UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO DE FIN DE GRADO

DISEÑO DE APLICACIÓN ANDROID PARA USOS DOMÓTICOS

AUTOR: RAÚL RUIZ IZQUIERDO

TUTOR: LUIS MENGIBAR POZO

FECHA: Leganés, Septiembre de 2015

1 AGRADECIMIENTOS

En primer lugar agradecer, por supuesto, a mis padres, Andrés y Feli, que gracias a todos su años de esfuerzo y trabajo me han permitido la realización de todos los estudios que me han permitido llegar hasta aquí, desde el colegio hasta la universidad, y que sin ellos y su apoyo y dedicación no hubiese podido llegar a ser la persona que soy en este momento.

En segundo lugar, a Luis la posibilidad de realizar el proyecto bajo su tutela, puesto que a lo largo de mis años de universidad curse diversas asignaturas impartidas por él y considero que es un profesor excelente y ha sido un placer realizar este proyecto con su ayuda y consejo.

Y, por último, a mis dos mejores amigas, Gemma y Lorena, que me han apoyado a lo largo de toda la realización de mi carrera y hasta la finalización de este proyecto.

Gracias a todos ellos porque sin su apoyo y ayuda no hubiese sido capaz de llegar a este momento.

2 ABSTRACT

This project describes the implementation of a graphical interface that could be used for managing and monitoring a home automation system using the data received from a wireless sensor block. For application implementation we are going to use Android programming and for communicating with the different elements that constitute the home automation system we are using Bluetooth technology, and more specific Bluetooth 4.0 also called Bluetooth Low Energy.

3 RESUMEN

En este proyecto se describe la implementación de una interfaz gráfica que podrá ser utilizada para la monitorización y control de un sistema domótico a través de los datos recibidos desde un bloque de sensores inalámbrico. Para la implementación de la aplicación emplearemos la programación en Android y para la comunicación con los diferentes sistemas que formen el sistema domótico se empleará la tecnología Bluetooth y, más concretamente, el Bluetooth 4.0 también denominado Bluetooth Low Energy.

4 GLOSARIO

Checksum: En español suma de verificación o de chequeo, es una función empleada para detectar cambios accidentales en una secuencia de datos con el objetivo de proteger la identidad de estos.

Dirección MAC: Siglas en inglés de "Dirección de Acceso al Medio" (Media Access Control) y es un identificador de 48 bits que corresponde de forma única a una tarjeta o dispositivo de red.

HashMap: Es una estructura de datos empleada para implementar un vector asociativo, que asocia claves con valores.

I^2C: Protocolo de comunicación serie half-duplex, con un bus de datos y otro de reloj, llamado Inter-Integrated Circuit (circuito inter-integrado).

Librería API: Procedente de las siglas en inglés de "Interfaz de Programación de Aplicaciones" (Application Programming Interface) y es el conjunto de subrutinas, funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software.

Método: En programación informática, un método consiste en una subrutina cuyo código es implementado en una clase y que puede pertenecer tanto a una clase como a un objeto.

PCB: Printed Circuit Board (Placa de Circuito Impreso, en español), es la superficie compuesta por canales, pistas o buses de material conductor laminadas sobre una base no conductora. Se usa para conectar y sostener componentes electrónicos.

SMBus: Bus de Administración del Sistema, es un subconjunto del protocolo I^2C.

SMD: Siglas en inglés de “Dispositivo de Montaje Superficial” (Surface Mounted Device), es un dispositivo empleado para el montaje de dispositivos electrónicos usando la “tecnología de montaje superficial” (SMT).

String: Cadena de caracteres formada por cualquier combinación finita de los caracteres disponible en el alfabeto inglés.

Thread: También denominado hilo, consiste en una unidad concurrente de ejecución de código. Todas las aplicaciones tiene como mínimo un Thread denominado Main Thread.

UUID: Identificador Único Universal (Universal Unique Identifier), es un número de 16 bytes (128 bits) usado en el proceso de construcción de software.

XML: Son las siglas en inglés de “Lenguaje de Marcas Extensible” (eXtensible Markup Language) y consiste en una forma de codificar un documento que, además de texto, incorpora etiquetas o marcas con información adicional de la estructura de texto o su presentación.

5 ÍNDICE DE CONTENIDOS

6 ÍNDICE DE FIGURAS

7 ÍNDICE DE TABLAS

8 INTRODUCCIÓN

En los últimos años, y especialmente en la última década, se producido un proceso de modernización, informatización y automatización de, un número cada vez mayor de aspectos de la vida. Este avance tecnológico se puede apreciar de especial manera en el auge de los móviles inteligentes o "Smartphone", los cuales aúnan en un solo dispositivo todas las funciones de un móvil además de la mayor parte de las de un ordenador. La necesidad de estos dispositivos se refleja en que en España los Smartphone suponen el 81% de los móviles en uso.

En este avance tecnológico también ha aumentado la instalación de sistemas de domótica en casa que permite el control de distintas partes de la vivienda. Este aspecto de la tecnología será el que nos ocupará a lo largo de este documento, en concreto nos centraremos en el control ambiental de la casa.

8.1 Objetivos

El objetivo final perseguido con la realización de este proyecto, es el desarrollo de una aplicación para dispositivos móviles que permita la gestión de un sistema domótico empleando para ello la tecnología inalámbrica Bluetooth.

Para lograr el cumplimiento del objetivo final del proyecto se ha dividido en dos objetivos principales, uno para cada una de las secciones de la aplicación. Dichos objetivos se detallan a continuación:

- Monitorización de los datos de temperatura y humedad proporcionados por los sensores del dispositivo *SensorTag*. Para el cumplimiento de este objetivo se han planteado dos objetivos a menor escala:
    - Establecimiento de la conexión con el dispositivo *SensorTag* a través de Bluetooth Low Energy.
    - Lectura y visualización en la pantalla de los datos proporcionados por el sensor escogido y actualización en tiempo real de los datos mostrados.

- Comunicación con un dispositivo controlador de un sistema domótico. Este objetivo también se ha dividido en dos de menor escala:
    - Conexión con el dispositivo encargado de controlar el sistema domótico.
    - Transmisión de los datos de configuración establecidos por el usuario de la aplicación.

8.2 Motivación del proyecto

Existen varios motivos por los cuales se ha decidido realizar este proyecto. En primer lugar, por tener la posibilidad de realizar un proyecto que pueda ser empleado en una tecnología que se encuentra en auge como es la domótica y que además sea de uso simple para que cualquier usuario pueda utilizarla sin necesidad de tener unos conocimientos avanzados en tecnología.

Así mismo, otro aliciente esencial para el desarrollo de este proyecto ha sido la posibilidad de aprender un nuevo lenguaje de programación, Java, aplicado a la plataforma Android, que aportará un valor añadido en una futura búsqueda de trabajo. Además, la flexibilidad ofrecida por la plataforma Android, facilita en gran medida el desarrollo de proyecto que están enfocados en el control de sistemas, puesto que no será necesario el desarrollo de un nuevo dispositivo electrónico de control, ya que es papel lo ejercerá un Smartphone o una Tablet, que son dispositivos muy extendidos en la actualidad.

8.3 Descripción del sistema

El sistema planteado para la realización del proyecto constará de partes distintas:

- Por un parte tendremos el conjunto de los sensores que se encontrarán en un dispositivo llamado *SensorTag*, diseñado por el fabricante de circuitos integrados Texas Instruments. Este dispositivo cuenta con un emisor de Bluetooth Low Energy que recibe y envía por medio de dicha tecnología los datos de los siguientes sensores:
    - Sensor de temperatura.

- Sensor de humedad.
- Sensor de presión.
- Acelerómetro.
- Magnetómetro.
- Giroscopio.

De estos seis sensores, para nuestra aplicación, emplearemos únicamente los sensores de temperatura y humedad, las características de los cuales serán descritos más adelante.

- Por otra parte tendremos una aplicación Android para Smartphone que cuenten con este sistema operativo. A partir de dicha aplicación podremos leer los datos recogidos de los sensores y en función de ellos programar y configurar el sistema domótico al que se encuentre enlazado la aplicación y que podrá ser escogido por el usuario.

9 ESTADO DEL ARTE

A lo largo de los siguientes apartados se dará una visión general del estado actual en el que se encuentra la tecnología y las tecnologías domóticas más empleadas en la actualidad.

9.1 ¿Qué es la domótica?

El término domótica proviene de los términos *domus,* palabra del latín que significa casa, e informática, por lo tanto el significado de domótica es la informatización o automatización de la casa.

La domótica consiste en todos los sistemas de automatización, gestión de la energía y seguridad los cuales se pueden aplicar a las viviendas y edificios. Para la implementación de un sistema domótico serán necesarios sensores, los cuales proporcionarán la información de la casa, y actuadores, que serán los encargados de realizar las acciones impuestas por el sistema o por el usuario. Entre los sensores y los actuadores se encuentra el sistema encargado de procesar la información para conseguir comodidad, gestión de energía eficiente o proteger a las personas, animales o bienes.

Un sistema domótico proporciona gran cantidad de servicios al usuario pero se pueden agrupar en cinco aspectos principales **[1]**:

- **Ahorro energético:** el objetivo es optimizar el uso de los sistemas de climatización y el consumo eléctrico de la casa. Por medio del uso de distintos sensores realizaremos una gestión más eficiente de la energía que finalmente acabará traduciéndose en un ahorro económico.
- **Confort:** El uso de un sistema domótico permite al usuario gozar de numerosas comodidades en el hogar, ya que le permite realizar la programación de escenas, la automatización de distintas actividades como puede ser subir o bajar las persianas, aparte de otras muchas opciones más que le permitiría el sistema al usuario.

- **Seguridad:** Permite la gestión de un sistema de seguridad encargada de la protección tanto de las personas como de los bienes de materiales, empleando para ella alarmas técnicas (incendios, fugas de gas, etc.) y alarmas de protección personal (simulación de presencia, detección de intrusos, etc.).

- **Comunicaciones:** Toda esta monitorización del sistema se puede llevar a cabo a distancia de tal manera que se pueden conocer y modificar el estado de los datos sin estar presente en casa por medio de la tecnología de redes inalámbricas como el 3G.

A lo largo de los siguientes apartados se dará una visión general del estado actual en el que se encuentra la tecnología y las tecnologías domóticas más empleadas en la actualidad.

9.2 Historia de la domótica

El inicio de la domótica se da en la década de los setenta, cuando comienzan a aparecer los primeros sistemas de automatización en edificios, aunque no fue hasta la década de los ochenta cuando se empezaron a usar los sistemas integrados a nivel comercial que posteriormente se desarrollaron para su uso en instalaciones domésticas urbanas, consiguiendo así integrar los sistemas eléctrico y electrónico **[2]**.

El desarrollo de la tecnología informática, que se produce sobre todo en países vanguardistas, permite integrar en los edificios el SCE o Sistema de Cableado Estructurado dando lugar a los edificios “inteligentes” ya que ponen a disposición del usuario todos sus automatismos disponibles. Estas primeras instalaciones estaban regidas por medio del sistema X-10, sobre el cual entraremos en detalle más adelante.

Con el avance de la tecnología se pudieron resolver los fallos que tenían inicialmente los sistemas domóticos ya que se pudo empezar a integrar eficientemente todos los sistemas y dispositivos tecnológicos presentes en el hogar, aunque no fue hasta finales de la década de los ochenta cuando la tecnología domótica comenzó a aplicarse a las viviendas particulares.

Ya en la actualidad, el aumento del mercado de esta tecnología ha permitido el desarrollo de nuevos sistemas de integración domótico como puede ser el sistema EIB/KNX, en el cual profundizaremos a lo largo de este documento.

A continuación explicaremos en más detalle los sistemas más empleados actualmente en la domótica.

9.3 Sistema X-10

El sistema X-10 es un estándar basado en la transmisión de corrientes portadoras, el cual es uno de los más antiguos de los que actualmente se están utilizando. Este sistema fue desarrollado por la empresa Pico Electronics en el año 1978 y en la actualidad todavía es comercializado en todo el mundo, pero el mercado mayoritario se encuentra en Estados Unidos **[3] [4]**.

La transmisión de corrientes portadoras consiste en un método de transmisión de datos a través de la red eléctrica de baja tensión. Para realizar este tipo de comunicación no es necesaria la instalación de una nueva red para la interconexión de los dispositivos, puesto que la transmisión de los datos puede realizarse a través de la instalación ya presente en el edificio. Además, puede trabajar en tanto en redes alternas monofásicas como trifásicas. Actualmente el desarrollo de la radiofrecuencia supone una alternativa a las redes eléctricas para el sistema X-10.

Pese a haber sido desarrollado por Pico Electronics, el sistema X-10 puede ser producido y comercializado por cualquier empresa que lo desee, aunque deberá emplear los circuitos electrónicos desarrollados en su momento por Pico Electronics.

Las características del sistema X-10 son las siguientes:

- Se trata de un sistema configurable, descentralizado y no programable.
- Fácilmente instalable.
- Fácilmente manejable para el usuario.
- Flexible y ampliable.
- Con un coste muy reducido.

9.3.1 Módulos del sistema X-10

Existen tres tipos distintos de módulos X-10:

- Módulos que únicamente reciben órdenes.
- Módulos que únicamente dan órdenes
- Módulos que pueden realizar ambas acciones

Para la identificación de cada módulo se emplea un sistema de letras y números en el cual, el "Código Casa" se representa por medio de una letra entre la A y la P, y el "Código Unidad" se representa por medio de un número comprendido entre 1 y 16. De esta manera un módulo podrá tener 256 direcciones distintas que se configuraran por hardware. En la Figura 1 se puede observar un módulo que formaría parte de un sistema X-10.

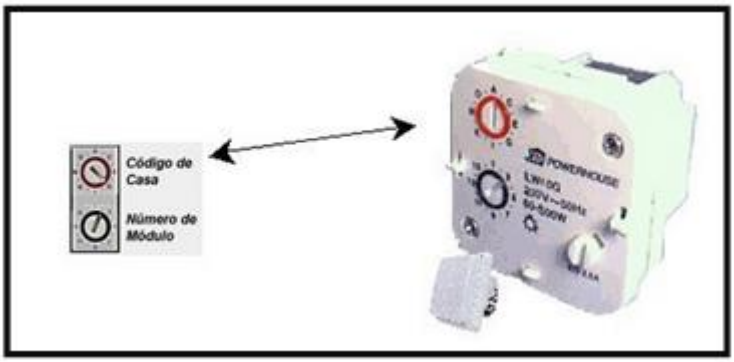

Figura 1. Módulo del sistema X-10 **[3]**

Se pueden configurar los receptores para poder recibir órdenes de distintos transmisores, además, también podremos asignar la misma dirección a diferentes receptores para poder recibir órdenes de un solo transmisor.

9.3.2 Transmisión de datos

Para realizar la transmisión de las órdenes entre dos módulos es necesario seguir una secuencia concreta. En primer lugar se debe transmitir la dirección del módulo con el que se desea comunicar (Código Casa - Código Unidad) y, a continuación, la instrucción a realizar por el receptor de la orden (Código Casa - Código Función). Por ejemplo, las instrucciones necesarias para activar y desactivar el propio módulo:

- Activación: C-3 + C-ON.
- Desactivación: C-3 + C-OFF.

La transmisión de los datos es síncrona con el paso por cero de la corriente alterna, que es controlado con un optoacoplador. La codificación en binario se realiza por medio de pulsos de 120 kHz dentro de la onda alterna de 50 Hz en Europa o 60 Hz en Estados Unidos, de tal manera, que el "1" es representado por la presencia de ese pulso de 120 kHz durante un milisegundo y el "0" lo representaría la ausencia de dicho pulso. Cuando el sistema es trifásico la transmisión se realiza tres veces, una por cada una de las fases. En la Figura 2 se muestra como se produce la transmisión de datos.

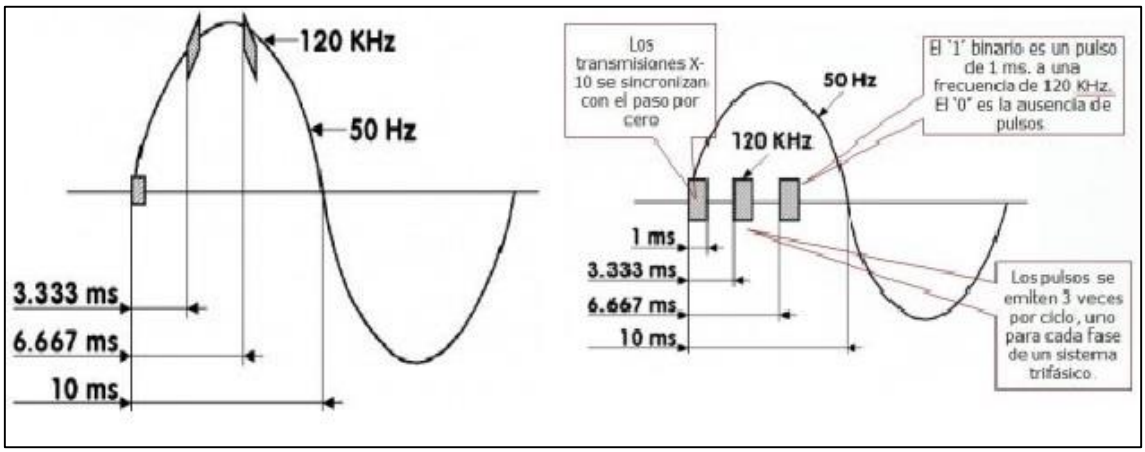

Figura 2. Transmisión de datos

Para la transmisión completa de un código serán necesarios once ciclos de corriente los cuales quedarían divididos de la siguiente manera:

- Los dos primeros ciclos representan el Código Inicio, que siempre es 1110 y que no cumple la condición de complementariedad.
- Los cuatro siguientes representa el Código Casa.
- Los cinco restantes representan el Código Unidad o el Código Función. Estos códigos son de cuatro bits, por lo tanto el quinto y último se emplea para que el receptor sepa si es Código Unidad (0) o Código Función (1).

Los bits se transmiten como parejas de bits complementarios empleando dos pasos por cero para cada bit, excepto, como ya hemos mencionado antes, el Código Inicio, es

decir, un cero quedaría representado por la secuencia 0-1 y un uno por la secuencia 1-0. En la Figura 3 se observa el envío de una única trama.

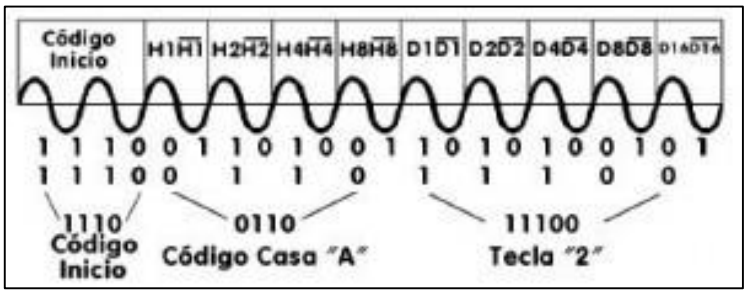

Figura 3. Envío de una trama

La secuencia completa de once ciclos se denomina trama y se realiza dos veces consecutivas (Figura 4), para una mayor fiabilidad del sistema, separando con tres ciclos de corriente el primer envío de la trama del segundo. La orden solo se realizará una vez se haya recibido la trama de confirmación, es decir, la segunda.

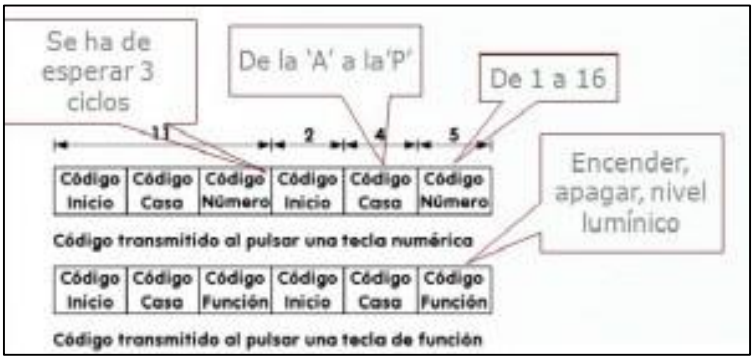

Figura 4. Doble envío de una trama

9.4 EIB/KNX

Seguidamente se procederá con la explicación de otro de los sistemas más extendidos actualmente para los sistemas domóticos, dándonos así una visión general de cómo se encuentra en la actualidad la tecnología de la domótica **[4]**.

9.4.1 Asociación KNX

La asociación Konnex, abreviada KNX, nace tras la creación del estándar KNX, en el año 1999. Tiene su sede en Bruselas y está formada por la fusión de las tres grandes asociaciones europeas de aplicaciones domóticas:

- BatiBus Club Internacional (BCI).

- European Installation Bus Association (EIBA).
- European Home System Association (EHSA).

La asociación KNX aúna en un solo protocolo los tres tipos de configuraciones diferentes de los sistemas (A-MODE, E-MODE, S-MODE), además de los diferentes medio físicos, para así poder asegurar una relación costes-prestaciones que sea adecuada para cualquier tipo de edificio o aplicación.

El objetivo que perseguía la creación de esta asociación era la unificación del estándar de comunicación y acordar con el CENELEC la elaboración de un conjunto de normas y reglas con el objeto de crear un protocolo común de comunicación que garantice la completa interacción entre los productos de los diferentes fabricantes. Por lo tanto, al igual que el sistema X-10, se trata de un sistema no propietario, es decir, no depende de una única marca o fabricante.

Cualquier dispositivo fabricado debe pasar una serie de ensayos normalizados para poder obtener la certificación que garantiza a los compradores que cumple es estándar KNX.

En 2003, la asociación CENELEC aceptó la tecnología KNX como un estándar domótico e inmótico. En ese mismo año, además, fue aceptado como estándar europeo, EN 50090, y en 2006 como estándar internacional con la norma ISO/IEC 14543-3.

Actualmente, la asociación KNX la integran más de 90 miembros distintos, compuestos por proveedores y fabricantes, 20000 compañías de instalación y 70 universidades técnicas.

9.4.2 Componentes del sistema

El sistema EIB/KNX lo componen una serie diversa de dispositivos los cuales son necesarios para el correcto funcionamiento de la instalación. Dichos dispositivos son el bus, los sensores, actuadores y dispositivos auxiliares (fuente de alimentación + filtro, interfaz serie y acopladores de línea o área). De los dispositivos listados únicamente describiremos los principales, que son los siguientes:

- **Bus:** Pueden ser dos dispositivos distintos:

- **Cable bus:** Formado por uno o dos pares trenzados, aislados y apantallados (2 x 2 x 0.8).
    - **Bus para perfil DIN:** Conecta los aparatos de la instalación para perfil DIN entre sí sin necesidad de cables. Se pueden conectar bien por medio de los terminales (bloques) o bien por medio de contactos a presión situados en la parte posterior del componente en contacto con el perfil.
    - **Sensores:** Son los dispositivos encargados del envío de los datos de la instalación al bus. Perciben los cambios en su estado y en función de ese cambio envía la información a los actuadores con una estructura de telegrama. Puede haber sensores de distintos tipos (pulsadores, termostatos, sensores de presencia, etc.).
- **Actuadores:** Son los encargados de recoger las órdenes procedentes de los sensores por medio del bus en forma de telegrama y transformas esas órdenes en acciones. También existen distintos tipos de actuadores (actuadores de conmutación, de regulación, de persianas, etc.).

9.4.3 Topología

La topología o arquitectura de un sistema KNX consiste en la estructura que adopta la red de comunicaciones mediante la cual se comunicarán cada uno de los componentes que están unidos por el bus de la instalación.

Esta estructura está dividida en dos partes, áreas y líneas. Las áreas están interconectadas entre ellas por medio de una línea dorsal principal y puede llegar a haber hasta 15 áreas distintas. A su vez, en cada una de estas áreas, existe una línea principal de la que pueden llegar a partir, también, hasta 15 líneas secundarias distintas.

Los distintos dispositivos que forma el sistema KNX estarán conectados a estas líneas secundarias, pudiendo llegar a tener cada línea hasta 256 dispositivos distintos, divididos en 4 segmentos de 64 dispositivos cada uno. De esta manera, si se emplease

toda la capacidad de cada área y de cada línea, se podrían conectar al sistema hasta 14400 dispositivos distintos.

Todas las líneas del sistema están conectadas entre sí por medio de Acopladores. Estos dispositivos actúan como aislante eléctrico entre las diferentes partes del sistema, de tal manera, que en el caso de producirse un fallo en alguno de los dispositivos conectados a las líneas no se pusiese en riesgo el funcionamiento de todo el sistema. Aparte de como aislante, los acopladores actúan también como filtro para los mensajes que van dirigidos a un dispositivo concreto, para así evitar que este mensaje se reparta por la línea completa. Los tipos de acopladores son los siguientes:

- **Acopladores de área:** Conectan la línea dorsal principal del sistema con la línea principal de cada área. Hay uno por área del sistema.
- **Acopladores de línea:** Conectan las líneas principales de cada área con cada una de las líneas secundarias. Hay uno por cada línea secundaria.

Además de los Acopladores, por cada línea de la instalación debe haber una fuente de alimentación de Bus que alimente a los dispositivos que se encuentren conectados en esa área. En la Figura 5 se puede observar como es la topología de este sistema.

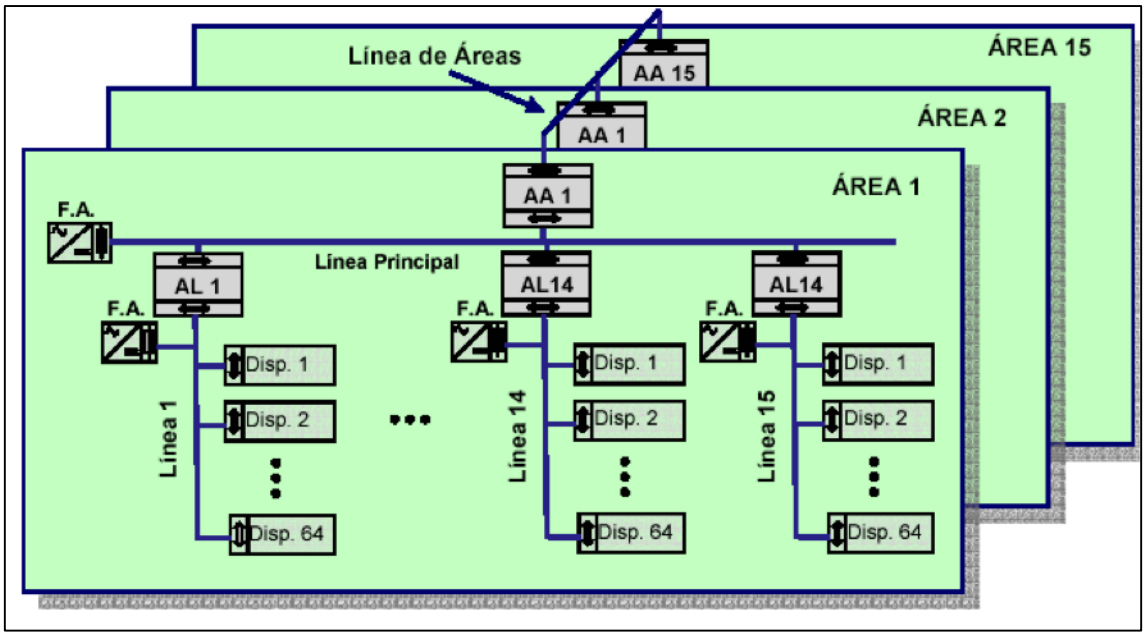

Figura 5. Topología de un sistema KNX **[5]**

Se pueden realizar distintas configuraciones del sistema. Pueden ser lineales, en estrella o en árbol. También se pueden realizar configuraciones híbridas. Las diferentes configuraciones posibles se muestran en la Figura 6.

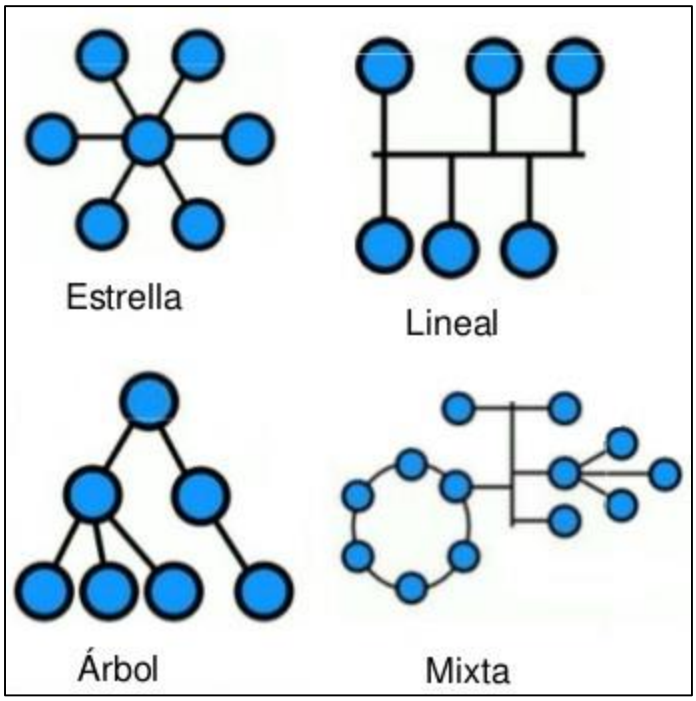

Figura 6. Configuraciones de una red KNX

La elección de la topología es libre, pero se deben respetar unas normas detalladas en la Tabla 1:

ELEMENTO A	DISTANCIA MÍN.	DISTANCIA MÁX.	ELEMENTO B
DISPOSITIVO	NO HAY	700	**DISPOSITIVO**
F. ALIMENTACIÓN	NO HAY	350	**DISPOSITIVO**
F. ALIMENTACIÓN	200	NO HAY	**F.ALIMENTACIÓN**

Tabla 1. Norma de distancia entre elementos del sistema

9.4.4 Direccionamiento

En un sistema EIBB/KNX cada uno de los dispositivos que forman parte de él está perfectamente localizado e identificados por medio de dos direcciones distintas, la dirección física y la dirección lógica o de grupo.

9.4.4.1 Dirección física

Esta dirección identifica cada componente de forma individual y única en la instalación, localizándolo en la estructura del sistema. La dirección sigue la secuencia Línea/Área/Componente. Para direccionar los dispositivos la dirección física consta de tres campos separados por puntos entre ellos y que completan un total de 16 bits divididos de la siguiente manera:

- **Área:** Los 4 primeros bits. Indica en cuál de las 15 áreas se encuentra. El 0 identifica la línea dorsal principal.
- **Línea:** Los 4 siguientes bits. Indica una de las 15 líneas del área. El 0 identifica la línea principal del área.
- **Componente:** Los 8 bits restantes. Identifica cada uno de los dispositivos que pueden estar conectados a la línea. El 0 identifica el acoplador de línea.

El uso de la dirección física es únicamente para tareas de diagnóstico y detección de errores o para modificación de la instalación. Por lo tanto durante el funcionamiento normal de la instalación esta dirección no tiene función alguna. En la Figura 7 se muestran las direcciones físicas de un sistema KNX.

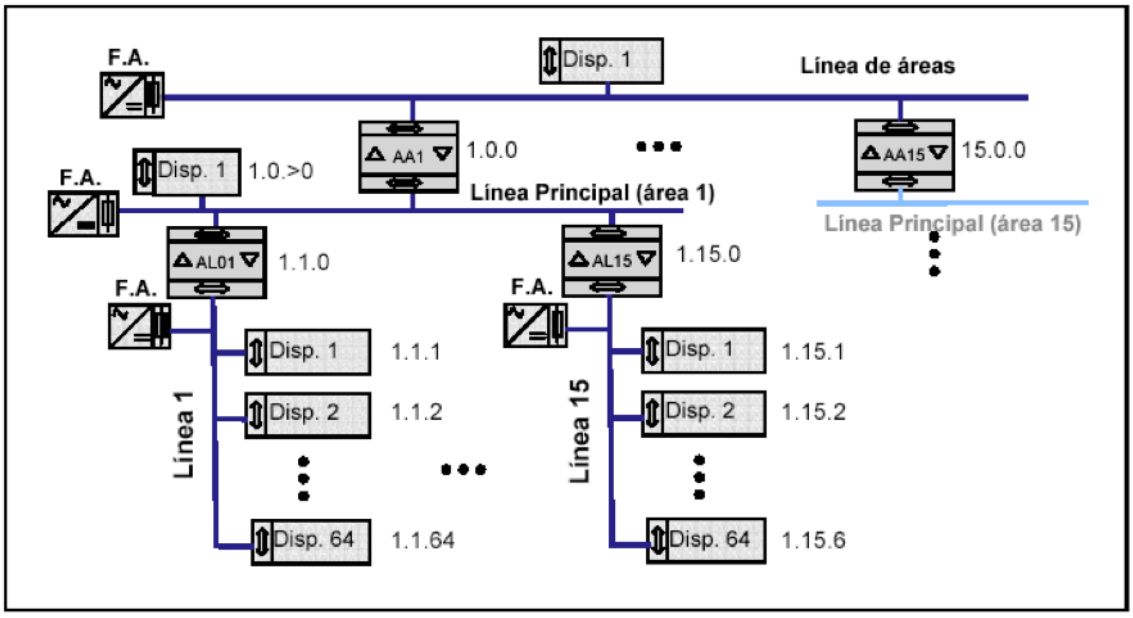

Figura 7. Direcciones físicas en un sistema KNX **[6]**

9.4.4.2 Dirección lógica o de grupo

Las direcciones de grupo se emplean para definir las funciones específicas del sistema, además de establecer las asociaciones entre los elementos de entradas del sistema, como son los sensores, y los de salida, es decir, los actuadores.

Para el direccionamiento lógico se pueden emplear dos tipos distintos de direcciones: de dos niveles o de tres niveles, dependiendo de las necesidades de jerarquización del sistema.

En un sistema KNX la asignación de direcciones de grupo es básica para poder asegurar el correcto funcionamiento de la instalación. Las direcciones de grupo (Figura 8) pueden asignarse a cualquier dispositivo de cualquier línea, con independencia de las direcciones físicas, pero cumpliendo las siguientes condiciones:

- Los sensores solo puede enviar una dirección de grupo, es decir, solo pueden tener asignada una dirección lógica.
- Diferentes actuadores pueden compartir la misma dirección de grupo, es decir, pueden responder todos ellos al mismo mensaje.
- Los actuadores pueden responder a más de una dirección lógica, esto es, pueden estar asociados a más de un sensor.

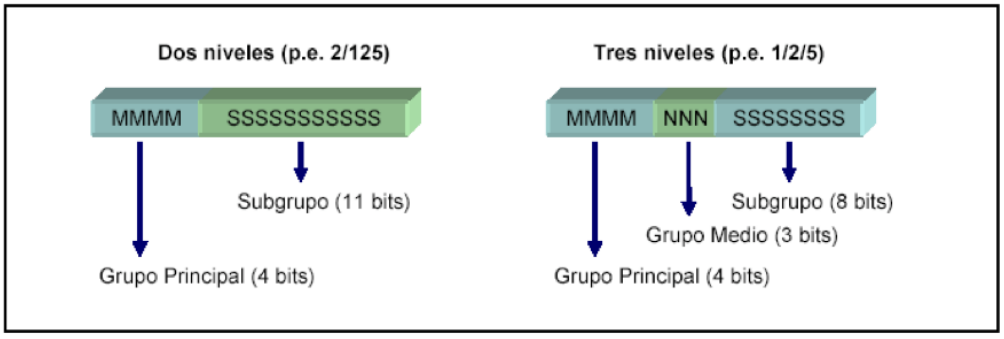

Figura 8. Dirección lógica o de grupo **[6]**

9.4.5 Acceso al medio

Una vez se ha producido el direccionamiento de los dispositivos, estos proceden en su acceso al medio, es decir, inyectan y recogen la información que necesitan en el bus.

Para este acceso al medio EIB KONNEX emplea el método de acceso de tipo CSMA/CA. La codificación se realiza de modo que el estado dominante, es decir, circula corriente, sea el del estado lógico '0', mientras que el estado lógico '1' será el estado recesivo, esto es, no circula corriente. El mecanismo empleado para la resolución de las colisiones será el siguiente **[6]**:

- El dispositivo comprueba el bus y solo comenzará con la transmisión cuando compruebe que este está libre.
- Mientras se produce el envío el dispositivo está continuamente escuchando y comparando los datos presentes en el bus con los que ha enviado. Pueden suceder dos cosas:
    - No se producen colisiones. En este caso el envío se produce sin contratiempos.
    - Se producen colisiones con los datos de otro equipo. En este otro caso tendrá prioridad los bits dominantes sobre los recesivos, por tanto, tendrán mayor prioridad aquellas tramas con mayor número de ceros a su inicio.

9.4.6 Procedimiento de transmisión de datos

La transmisión de un telegrama se produce cada vez que se produce un evento en el sistema. Por ejemplo, si se detectan niveles inadecuados de gas, se activa un detector de presencia, etc. Para realizar la comunicación el dispositivo emisor, que es el sensor, comprueba durante un tiempo t1 que el bus está disponible y una vez lo ha comprobado envía el telegrama. Si no se producen colisiones durante la transmisión, cuando esta finaliza espera nuevamente un tiempo t2 la recepción de la señal de reconocimiento o Acknowledgement. En el caso de que se reciba la señal de no reconocimiento o no se reciba nada, es decir, la transmisión sea incorrecta, vuelve a reintentar la comunicación hasta un máximo de tres veces.

La transmisión de los telegramas se realiza de modo asíncrono a una velocidad de 9600 baudios, esto es, 9.6 kbits/s y cada palabra de las que forman el telegrama están

compuestas por 1 bit de inicio, 8 bits de datos, 1 bit de paridad par, 1 bit de parada y 2 bits de espera hasta la siguiente palabra.

Con esta estructura la transmisión de una palabra supondrá un tiempo de 1.35 ms, con lo que el envío de un telegrama completo será de entre 20 y 40 ms, aunque la mayoría de las órdenes consisten en órdenes de marcha-paro, con lo que el tiempo tiende más a los 20 ms.

El telegrama es transmitido por medio del bus e incluye la información específica del evento ocurrido. Este está compuesto por siete campos diferenciados, seis de ellos son de control para asegurar una transmisión fiable y el campo restante formado por datos útiles con el comando a ejecutar. Los campos que formarían el telegrama serían los siguientes (Figura 9) **[7]**:

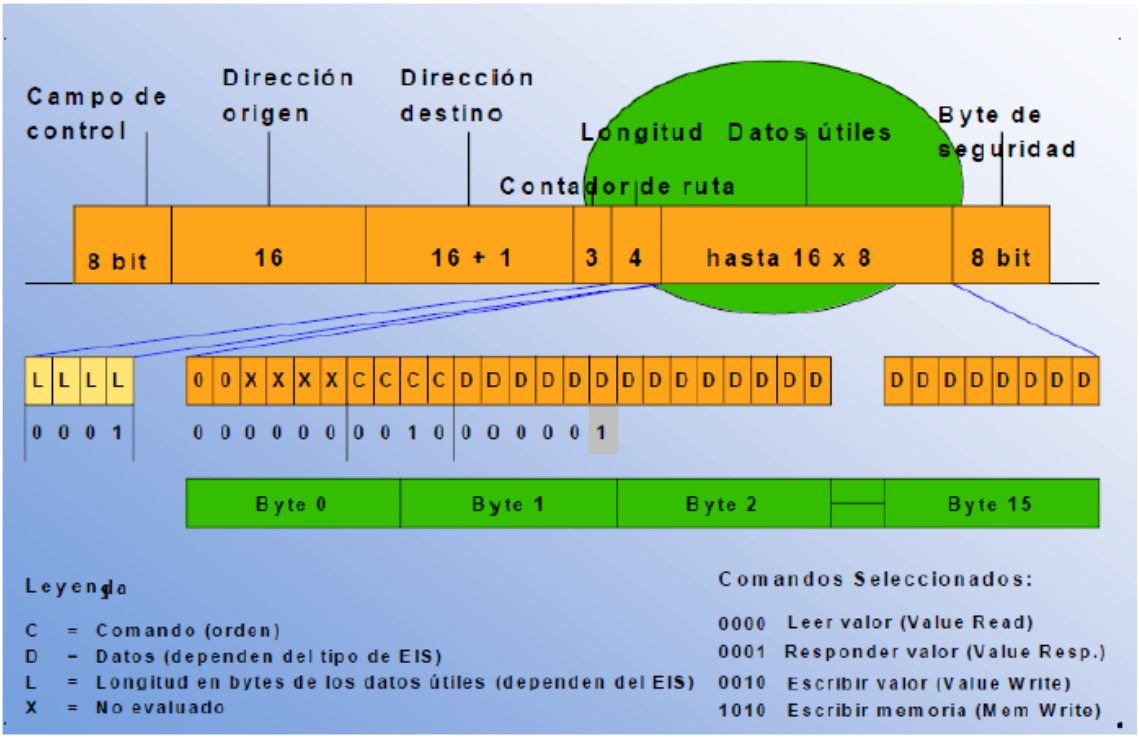

Figura 9. Estructura de un telegrama **[4]**

- **Campo de control:** Es un campo de 8 bits que contiene la prioridad del telegrama al ser enviado según el tipo de función. Incluye, además, un bit de repetición que se pone a '0' en caso de que se repita el envío por falta de reconocimiento y poder evitar la repetición de tareas ya ejecutadas.

- **Dirección de origen:** El dispositivo emisor transmite su dirección física, con la estructura explicada anteriormente, de modo que el emisor del telegrama sea conocido en las tareas de mantenimiento.

- **Dirección de destino:** La dirección de destino puede ser de dos tipos, en función del valor del MSB (Most Significant Bit, en español, Bit Más Significativo), que en este caso es el bit 17. Pude ocurrir:

    - El valor del MSB se '0'. En este caso indica que se trata de una dirección física y por lo tanto el telegrama se dirige únicamente a un dispositivo.

    - El valor del MSB es '1'. En este caso se trata de una dirección lógica y, por lo tanto, el telegrama se dirige a todos los mecanismos que deben escucharlo, que serán los que tengan dicha dirección lógica.

- **Campos de longitud e información útil:** En este campo se envían los datos necesarios para la ejecución de las órdenes y transmisión de valores. En el campo de longitud, 4 bits, se indica el número de bytes que compondrán el campo de datos siendo "0000" 1 byte y "1111" 16 bytes.

- **Campo de comprobación:** Este campo está formado por un byte obtenido de la realización del cálculo de la paridad longitudinal par (LRC2) de todos bytes incluidos anteriormente en el telegrama. Cuando el dispositivo recibe el telegrama comprueba si es correcto por medio de este byte. De esta manera si el envío se produce correctamente enviará un reconocimiento (ACK), en caso contrario, se enviará un no reconocimiento (NAK) para que el emisor repita el envío del telegrama.

9.4.7 Programación de la instalación

La última fase de la realización de una instalación de un sistema EIB/KNX es la programación de la misma. Para realizarla, habitualmente, se emplea un ordenador conectado al bus del sistema por medio de un puerto USB. En la fase de programación se establecen las direcciones físicas de los dispositivos, se cargan los programas de aplicación en todos los componentes y se asignan las direcciones lógicas de los

mismos. La programación se realiza empleando herramientas informáticas específicas para este tipos de proyectos, como, por ejemplo, el ETS (Engineering Tools Software).

Al realizar la programación de la instalación podremos obtener las bases de datos de los componentes EIB del sistema a través de las páginas web de los fabricantes, para después cargarlas en el software ETS.

9.4.8 Ventajas e inconvenientes

Las ventajas de este sistema son numerosas ya que la domótica ofrece gran cantidad de posibilidades y cualquiera puede desarrollar algo nuevo gracias al gran avance que está teniendo la tecnología en la actualidad.

Las ventajas más destacadas de los sistemas EIB/KNX serían **[8]**:

- Es un sistema descentralizado, es decir, no requiere centralización de los sistemas ni a físico ni lógico.
- Se puede indicar al sistema como actuar tras un fallo de corriente, de tal manera que evitaremos comportamientos imprevisibles del sistema tras un corte de luz en la vivienda o el edificio.
- Al contrario de lo que ocurre en las instalaciones tradicionales, puede controlar, vigilar y comunicar todas las funciones de servicio y su desarrollo con una única línea común. Así, puede dirigir la línea de energía sin desvíos, directamente hasta el aparato consumidor.
- Permite la ampliación y modificación de la instalación con facilidad. Además tiene una rápida adaptabilidad ante los cambios de uso o reorganización del espacio.
- Se produce un incremento en la seguridad.
- Se realiza un consumo económico y racional de la energía en el funcionamiento de edificios.
- Tiene un mayor grado de confort.

- Se trata de un sistema abierto, esto es, las especificaciones del sistema están disponibles para todo el mundo.

Pese a todas estas ventajas, este sistema también tiene sus inconvenientes:

- Se trata de un sistema que carece de redundancia, es decir, si se produjese un fallo en la línea perderíamos todos los dispositivos asociados a esta. En el peor de los casos, si el fallo se produjese en la línea principal, perderíamos toda la instalación. A este problema se le podría poner como solución el empleo de un doble cableado que en caso de fallo en la línea se conectaría para poder continuar empleando la instalación.
- En caso de que se produjese una saturación del bus se podrían llegar a tener hasta dos segundos de retraso, con lo que podría ocurrir que se realizasen los 3 intentos que realiza el emisor sin producirse el envío y sin registrar el error.
- No se pueden realizar simulaciones del sistema puesto que el software no lo permite.
- Se trata de un sistema con un precio muy elevado.
- Para realizar una instalación con este sistema es necesario una gran inversión inicial, por lo tanto esto está frenando su uso.

10 BLUETOOTH

A lo largo de este apartado se dará una visión general de lo que es el Bluetooth, sus principales características y como es su funcionamiento. La programación relacionada con esta tecnología se explicará en el apartado relacionado con la implementación de la aplicación.

10.1 Introducción

Bluetooth es una tecnología de Red de Área Personal Inalámbrica, abreviada por sus siglas en inglés WPAN (Wireless Personal Area Network), que se utiliza para conectar dispositivos entre sí sin necesidad de una conexión por cable. A diferencia de otras tecnologías inalámbricas, como la de comunicación por infrarrojos (IRDA), el Bluetooth no necesita visualización directa entre los dos dispositivos para poder realizar la comunicación **[9]**.

Esta tecnología se diseñó principalmente, para realizar conectar dispositivos, equipos y PDAs entre sí, empleando circuitos de radio de bajo coste, a distancias reducidas y con un gasto de energía mínimo y todo ello evitando el uso de cables. En la actualidad su uso más extendido es en los teléfonos móviles y especialmente para la conexión de estos con dispositivos “manos libres”, como pueden ser los auriculares.

El origen de la tecnología Bluetooth se encuentra en el año 1994 y fue desarrollada por la empresa Ericsson. Años más tarde, en 1998, se crearía un grupo denominado Bluetooth Special Interest Grup (Bluetooth SIG), que comprendía a más de 200 compañías internacionales. El objetivo de este grupo era desarrollar las especificaciones para el Bluetooth 1.0, que finalmente fueron publicadas en el año 1999. Desde entonces la tecnología del Bluetooth a continuado desarrollándose a lo largo de los años hasta llegar hasta nuestros días y al Bluetooth 4.0 o Bluetooth Low Energy tal y como se muestra en la Figura 10.

Como anécdota curiosa, cabe destacar que el nombre “Bluetooth” proviene del rey Harald I (910 - 986), que era apodado Harald I Blåtand (en inglés "blue-toothed"), y que fue quién logró la unificación de Suecia y Noruega, e introdujo el cristianismo en Escandinavia.

Version	Date	Key features
1.0	5 July 1999	Draft version
1.0a	23 July 1999	First published version
1.0b	December 1999	Bug fixes
1.0b +CE	November 2000	Critical errata added
1.1	February 2001	The first solid release and the basis of growth. This version was ratified by the IEEE as the 802.15.1–2002 standard.
1.2	November 2003	Included adaptive frequency hopping. Added eSCO for better voice performance. Ratified by the IEEE as 802.15.1–2005, although no subsequent releases have been added to it, leaving the 802.15.1 standard as an orphaned specification.
2.0 + EDR	November 2004	Added enhanced data transfer rate to increase throughput to 3.0 Mbps
2.1 + EDR	July 2007	Added secure simple pairing to improve security and usability
3.0 + HS	April 2009	Added 802.11 as a high-speed channel, boosting rates to 10 Mbps and above
4.0 + HS	December 2009	Includes Bluetooth low energy

Figura 10. Versiones de la tecnología Bluetooth **[10]**

A continuación procederemos con la explicación de las principales características que posee la tecnología Bluetooth.

10.2 Principales características

El sistema Bluetooth puede llegar a transmitir a velocidades de hasta 1 Mbps, que correspondería a 1600 canales en una comunicación full-dúplex. La potencia y, en consecuencia, el alcance de la emisión depende, además, del tipo de transmisor empleado. Existen tres tipos distintos de transmisores cuyas características se detallan en la Tabla 2:

CLASE	POTENCIA(Pérdida de señal)	ALCANCE
I	100 mW (20 dBm)	100 m
II	2.5 mW (4 dBm)	15-20 m
III	1 mW (0 dBm)	10 m

Tabla 2. Clases de transmisores Bluetooth **[9]**

Para la transmisión de los datos el Bluetooth emplea ondas de radio, empleando la banda de frecuencia de 2.4 GHz, por lo tanto no es necesaria el contacto visual entre los dos dispositivos que se están comunicando, pudiendo producirse la comunicación incluso con la presencia de muros entre ambos dispositivos. Además, para realizar la comunicación no es necesaria siquiera la presencia de un usuario, puesto que los dispositivos Bluetooth pueden detectarse los unos a los otros siempre y cuando estén dentro del radio del alcance del transmisor.

En el siguiente apartado se desarrollará el principio de funcionamiento en el que se basa esta tecnología.

10.3 Principio de funcionamiento

El estándar Bluetooth emplea para realizar la comunicación el método Frecuency Hopping Spread Spectrum (FHSS). Dicho método consiste en dividir la banda de frecuencia comprendida entre 2.402 GHz y 2.480 GHz en 79 canales (denominados “saltos”), cada uno de ellos con ancho de banda de 1 MHz, para después empleara cada vez uno distinto para la transmisión de datos en una secuencia que deberá ser conocida tanto por el emisor como por el receptor **[11]**.

Por lo tanto, al realizarse el cambio de canales a una frecuencia de 1600 cambios por segundo, se evitan las interferencias que pudiese haber con otras señales de radio.

Una vez finalizado el principio de funcionamiento se procederá con el principio de comunicación.

10.4 Principio de comunicación

El sistema Bluetooth se basa en el modo de operación maestro/esclavo. Para explicar cómo se produce la comunicación emplearemos el término "piconet", que hace referencia a la red formada por un dispositivo maestro y todos sus dispositivos esclavos. Una piconet puede estar formada por 2 a 8 dispositivos distintos, en el que uno siempre actuará de maestro y los restantes de esclavos, por tanto, un maestro puede llegar a conectarse simultáneamente hasta a 7 dispositivos distintos. Cada uno de los dispositivos presentes en una red piconet, mostrados en la Figura 11, posee una dirección lógica de 3 bits **[11]**.

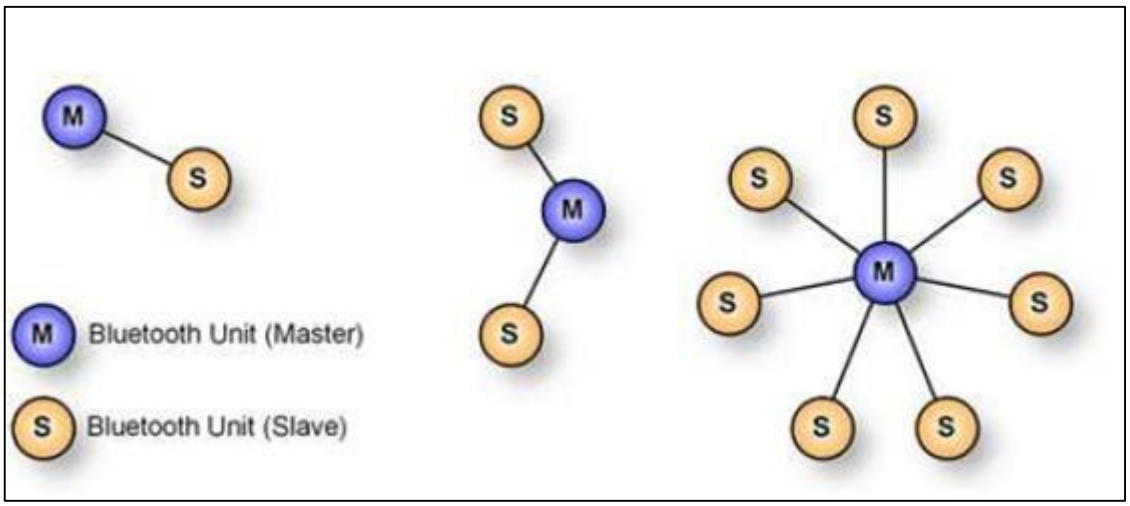

Figura 11. Red piconet

En realidad, instantáneamente, un maestro solo puede conectarse únicamente a un esclavo, por lo tanto, el maestro cambia la conexión de un esclavo a otro distinto a una frecuencia muy elevada dando la impresión de que está conectado con todos los dispositivos al mismo tiempo.

Además, es posible que varias redes piconet se conecten entre sí empleando ciertos dispositivos como puente entre ellas, cuando esto sucede la red que se forma se denomina “scatternet”, mostrada en la Figura 12.

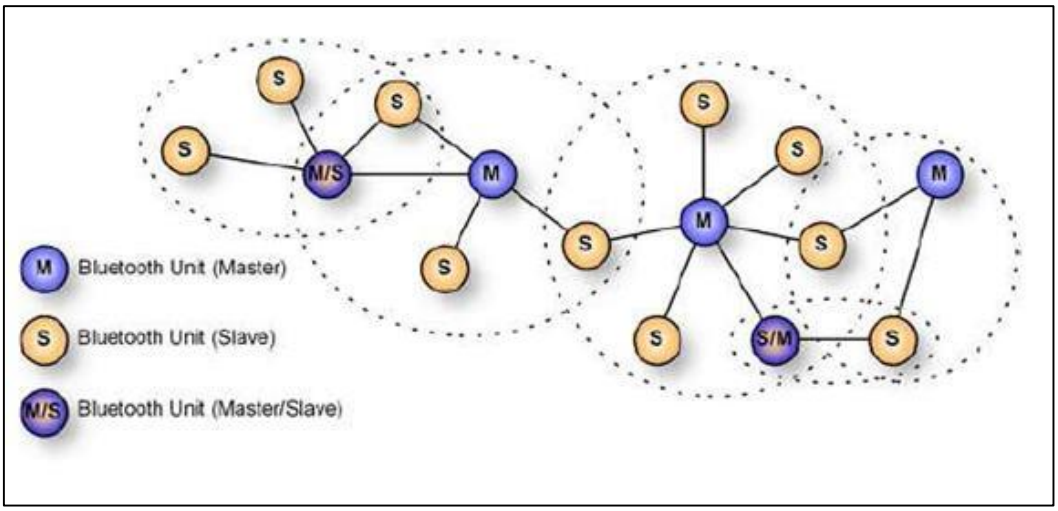

Figura 12. Red scatternet

Por último se desarrollará el proceso llevado a cabo para establecer una conexión por medio de Bluetooth.

11 SENSOR INALÁMBRICO

A lo largo de este apartado se desarrollará la explicación sobre todos los aspectos relacionados con los sensores que emplearemos para el desarrollo de la aplicación. El dispositivo escogido como bloque para los sensores es denominado *SensorTag*. Todos los datos referidos a los diferentes sensores y componentes que se incluyen en el documento han sido obtenidos de las hojas de características oficiales del fabricante de cada uno de ellos.

11.1 *SensorTag*

Un *SensorTag* (Figura 13) es un dispositivo, creado por la empresa de electrónica Texas Instruments, basado en la tecnología de transmisión de datos de manera inalámbrica. Se trata del único dispositivo del mercado diseñado para el desarrollo de aplicaciones para Smartphone y además ofrece una gran versatilidad a la hora de diseñar aplicaciones basadas en este dispositivo. Esta variedad de opciones que nos ofrece viene dada por el número de sensores de los que dispone. Los sensores de los cuales dispone un *SensorTag* son los siguientes:

- **Temperatura:** el sensor de temperatura es un "TMP006 *Contactless IR Temperature Sensor*", fabricado por la misma empresa que el *SensorTag*, Texas Instruments.
- **Humedad:** el sensor de humedad es un "SHT21 Humidity Sensor", fabricado por la empresa electrónica Sensirion.
- **Giroscopio:** el giroscopio es un "IMU-3000 Gyroscope", fabricado por la empresa Invenesense.
- **Acelerómetro:** el acelerómetro es un "KXT J9 Accelerometer", fabricado por Kionix.
- **Magnetómetro:** el magnetómetro es un "MAG3110 Magnetometer", el cual lo fabrica la empresa Freescale.

- **Presión:** el sensor de presión es un "T5400 (C953H) Barometric Pressure Sensor" fabricado por la empresa Epcos.

Figura 13. SensorTag

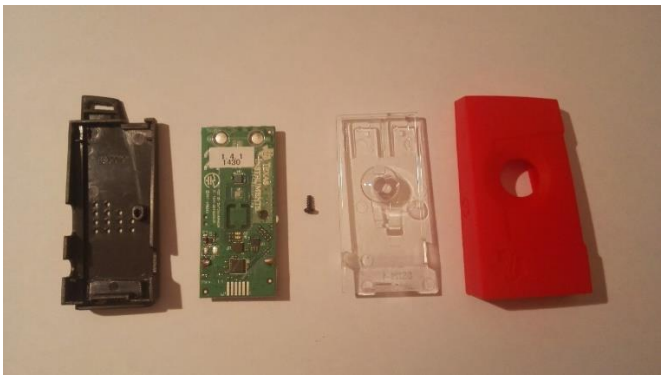

Figura 14. Diagrama de bloques del dispositivo

A pesar de que disponemos de seis sensores distintos para usar, en nuestra aplicación únicamente necesitaremos los de humedad y temperatura, por tanto explicaremos más detalladamente en apartados posteriores.

Además de los sensores ya mencionados, el *SensorTag* tiene otros dos componentes fundamentales para el correcto funcionamiento, los cuales también se explicarán en detalle en futuros apartados, tanto de la aplicación como del dispositivo en sí:

- **Transmisor Bluetooth Low Energy:** se encargará de enviar los datos al Smartphone. Está fabricado por la empresa Texas Instruments y es un "CC2541 Bluetooth Low Energy Radio SoC".

- **Convertidor DC/DC:** este último componente se encargará de la alimentación del dispositivo, es un "TPS62730 Ultra Low Power DC/DC Converter" fabricado por Texas Instruments.

Todos los sensores que equipa el dispositivo están diseñados para ser pequeños, de bajo consumo y SMD (*Surface Mount Device*). Todos los sensores se encuentran conectados a un mismo bus de datos por el que realizarán la comunicación, esta se realizará empleando el protocolo de comunicaciones I2C. Además, para conseguir que el consumo del dispositivo sea el mínimo, los sensores se encontraran inicialmente desactivados y deberán ser activados por software, del modo que explicaremos en el apartado sobre la interfaz gráfica. El diagrama de bloques del dispositivo *SensorTag* se muestra en la siguiente figura.

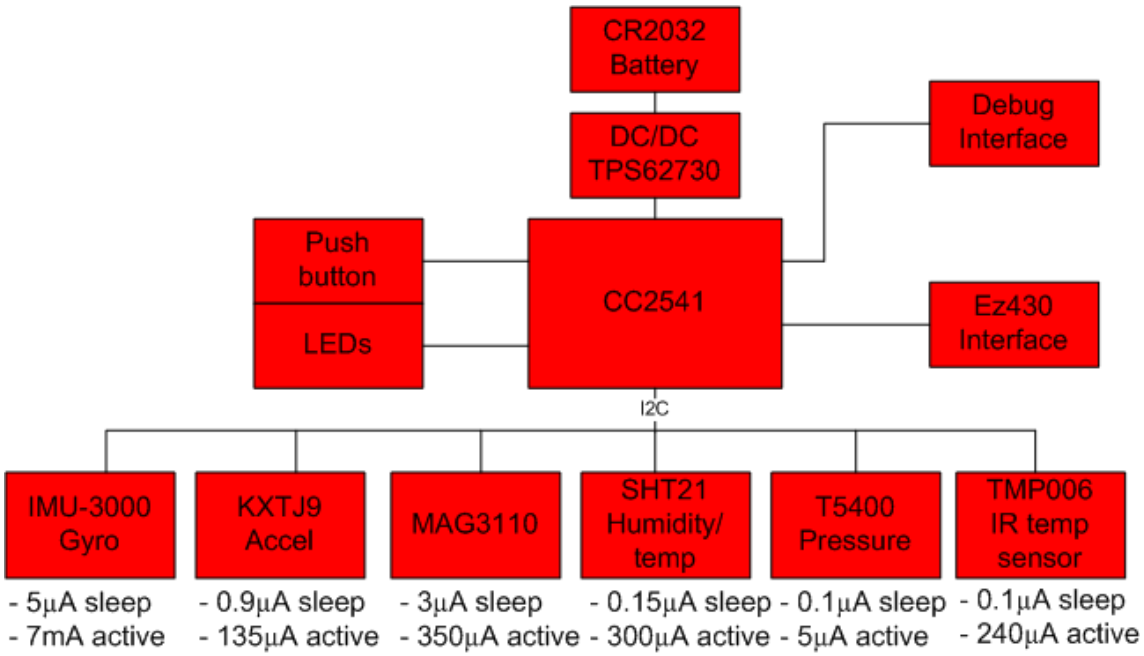

Figura 15. Diagrama de bloques del dispositivo **[12]**

La alimentación del dispositivo se realizará empleando una pila de botón de 3 voltios. En la figura **x** se muestra la distribución de los componentes en la placa de circuito impreso (PCB).

A continuación procederemos con la explicación detallada de las características y funcionamiento de los sensores que emplearemos para el desarrollo de nuestra aplicación, es decir, el sensor de humedad y temperatura.

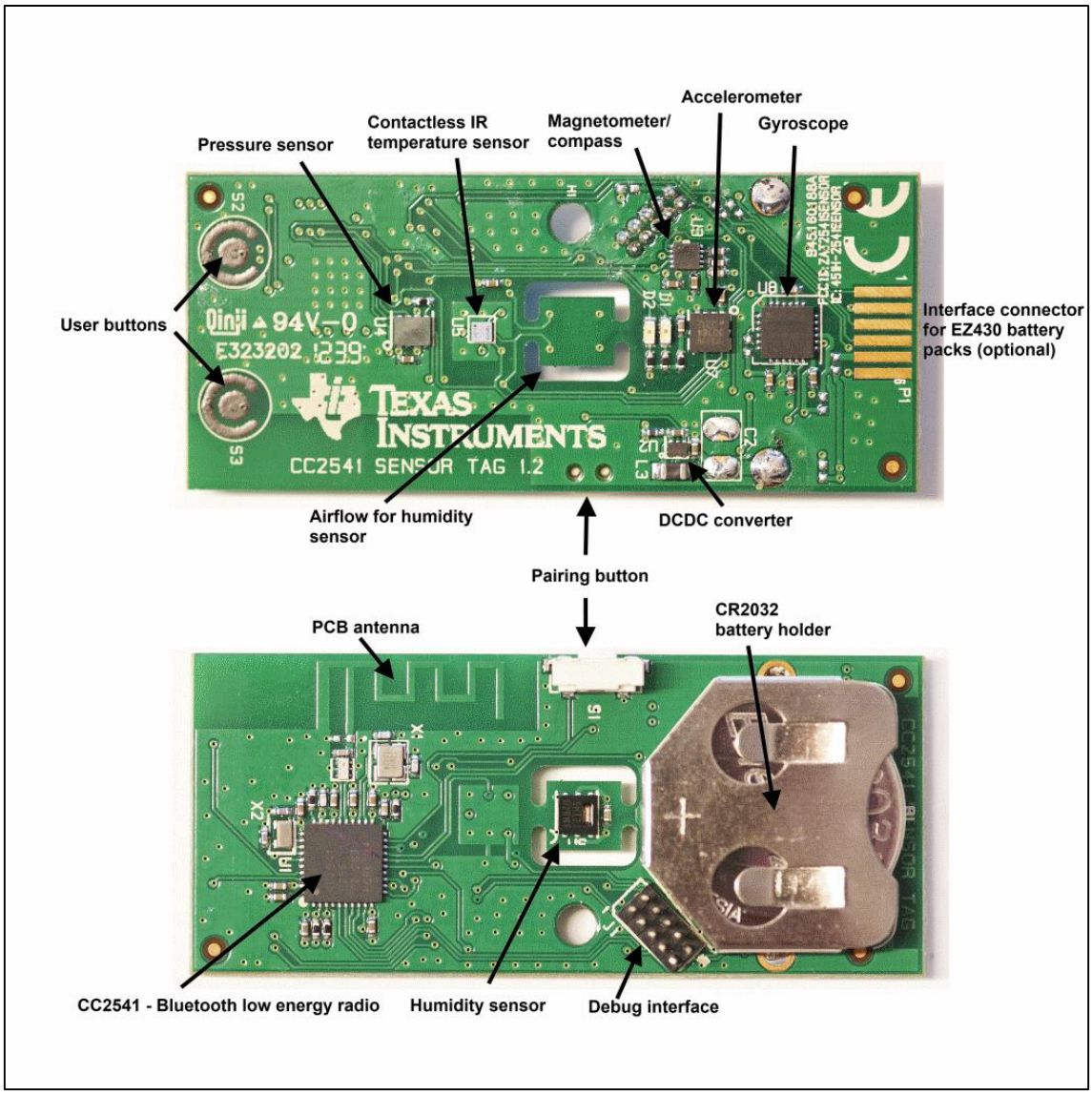

Figura 16. Distribución de los componentes del SensorTag *en la PCB* **[12]**

11.1.1 Sensor de temperatura

Como ya se ha indicado anteriormente, el sensor utilizado para la detección de la temperatura es el sensor fabricado por la empresa Texas Instruments "TMP006 *Contactless IR Temperature Sensor*". Este sensor esta optimizado para la toma los casos en los que la medida de la temperatura se realice sin contacto. Los datos captados por el sensor son transmitidos por medio de un interfaz serie de doble cable, es compatible con SMBus e I2C, este último, como ya se ha mencionado, es el utilizado para la comunicación con el transmisor Bluetooth. Para calcular la temperatura emplea

tanto la temperatura ambiente como los cambios de voltaje en el sensor. Así mismo, ambos datos son transmitidos por Bluetooth. En la figura **x** se muestra el diagrama de bloques del sensor.

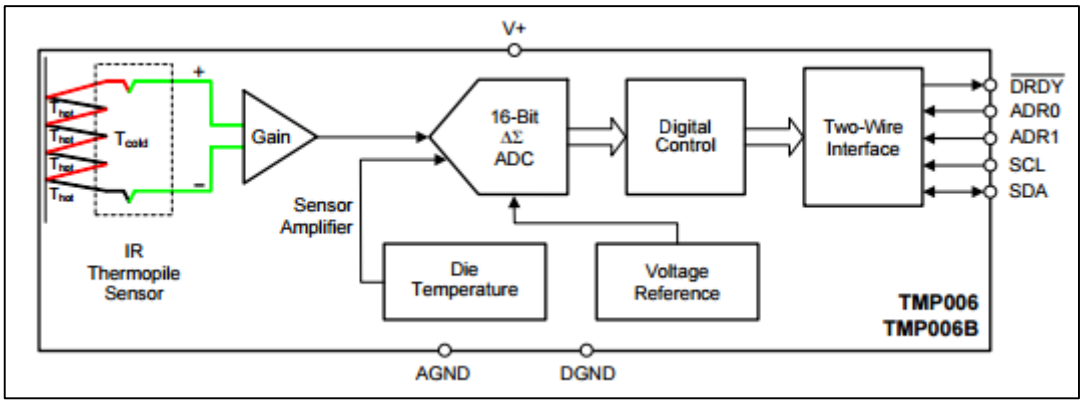

Figura 17. Diagrama de bloques del sensor de temperatura **[13]**

Las principales características del sensor son las siguientes:

- Termopila MEM (Microelectromecánica) integrada para la medida de temperatura sin contacto.
- Sensor de temperatura para la referencia de la unión fría (punto frío) de la termopila.
- Transmisión de datos vía serie. Compatible con I^2C y SMBus, a 3,3 V.
- Consumo reducido:
    - Alimentación: 2,2 V – 5,5 V.
    - Corriente (Activo): 240 µA.
    - Corriente (Inactivo): 1 µA.
- Compacto. 1,6 mm x 1,6 mm x 0,625 mm.
- Error: ±1ºC (0ºC – 60ºC)

En cuanto al modo de funcionamiento, el sensor utiliza el cambio que se produce en la diferencia de potencial de la termopila. Para ello, la termopila utiliza la diferencia de temperatura entre la unión caliente (punto caliente), $T_{HOT,}$ y la unión fría, $T_{COLD,}$ y genera una tensión proporcional a dicha diferencia de temperatura. Para la

temperatura fría se emplea la temperatura ambiente, y para la temperatura de la unión caliente se emplea la radiación absorbida por la unión caliente (figura **X**). La ecuación 1 **[13]** calcula la tensión de salida de la termopila:

$$V_{OUT} = C\,(T_{HOT} - T_{COLD}) \propto (T_{OBJ}^4 - T_{DIE}^4) \quad (1)$$

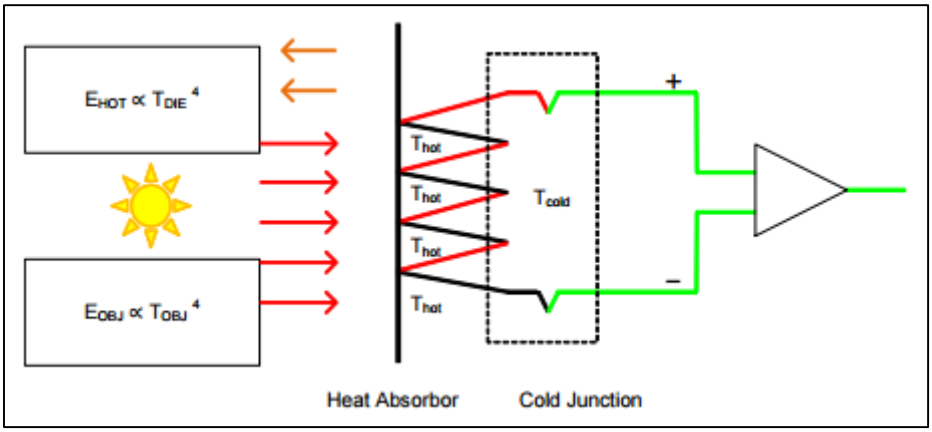

Figura 18. Funcionamiento de la termopila **[13]**

Esta tensión V_{OUT}, junto con la temperatura ambiente y la tensión de referencia, pasará por un convertidor analógico-digital (ADC) de 16-bit, que las convertirá en una señal digital que será enviada a través del bus serie al transmisor Bluetooth. Para obtener la temperatura, en Kelvin, del objeto enfocado por el sensor IR, serán necesarias las siguientes ecuaciones **[14]**, que serán las que implementemos en el código de la aplicación, donde $V_{OBJ} = V_{OUT}$.

La ecuación 2 representa la sensibilidad de la termopila:

$$S = S_0[1 + a_1(T_{DIE} - T_{REF}) + a_2(T_{DIE} - T_{REF})^2] \quad (2)$$

La siguiente ecuación obtiene el *offset* de la tensión:

$$V_{OS} = b_0 + b_1(T_{DIE} - T_{REF}) + b_2(T_{DIE} - T_{REF})^2 \quad (3)$$

En la ecuación 4 se describe la función de la tensión.

$$f(V_{OBJ}) = (V_{OBJ} - V_{OS}) + c2(V_{OBJ} - V_{OS})^2 \quad (4)$$

Con la última ecuación obtendremos la temperatura del objeto.

$$T_{OBJ} = \sqrt[4]{T_{DIE} + \left(\frac{f(V_{OBJ})}{S}\right)} \quad (5)$$

Los valores de los coeficientes empleados en el cálculo de la temperatura se detallan a continuación:

- V_{OBJ}: Tensión de salida de la termopila enviada vía serie
- T_{DIE}: Temperatura ambiente enviada vía serie.
- S_0: Factor de calibración, en nuestro caso, 5,593 x 10^{-14} .
- a_1: 1,75 x 10^{-3}.
- a_2: -1,678 x 10^{-5}.
- T_{REF}: 298,15 K (25 ºC).
- b_0: -2,94 x 10^{-5}.
- b_1: -5,7 x 10^{-7}.
- b_2: 4,63 x 10^{-9}.
- c_2: 13,4.

Con esto finaliza la explicación sobre el sensor de temperatura incorporado en el dispositivo *SensorTag*, seguidamente procederemos con el sensor de humedad.

11.1.2 Sensor de humedad

En este caso, el sensor encargado de la toma de medidas de la humedad es el fabricado por la empresa Sensirion y es denominado “SHT21 Humidity Sensor”. Pese a que este sensor permite también la medida de la temperatura, únicamente se emplea el dato de la humedad, aunque incluiremos la temperatura en la explicación que sigue. El bloque de este sensor está formado por los siguientes componentes:

- Un sensor de humedad capacitivo, que tomará medidas según el cambio capacidad producido en el mismo.
- Un circuito integrado digital y analógico especializado para este sensor.
- Por último, un sensor de temperatura de banda prohibida.

Todos estos componentes le proporcionan a sensor un rendimiento muy elevado con un consumo altamente reducido. Además, la sensibilidad del sensor puede ser cambiada, entre 8 bit y 12 bit para la humedad y entre 12 bit y 14 bit para la temperatura, e incluye un *checksum* para mejorar la fiabilidad de la comunicación.

Así mismo, aparte de las características ya mencionadas en este apartado, el sensor de humedad tiene también las siguientes **[16]**:

- Medidas: 3mm x 3mm x 1,1mm.
- Comunicación: Protocolo I^2C.
- Alimentación: 2,1 V – 3,6 V.
- Consumo de energía: 3,2 µW (Con una sensibilidad de 8 bit y una medida por segundo).
- Rango de medida de humedad: 0% - 100%.
- Rango de medida de temperatura: -40ºC – 135 ºC.
- Error: ±2%

Finalmente explicaremos las ecuaciones necesarias para obtener la humedad, que implementaremos posteriormente en el código de nuestra aplicación, y la temperatura a partir de los datos recibidos del sensor **[16]**.

La ecuación 6 la emplearemos para calcular la humedad en porcentaje, donde S_{RH} es la señal de la humedad recibida del sensor:

$$RH = -6 + 125(S_{RH}/2^{16}) \quad (6)$$

La siguiente es la ecuación necesaria para obtener la temperatura en ºC, donde S_T es la seña de la temperatura recibida del sensor:

$$T = -46{,}85 + 175{,}72(S_T/2^{16}) \quad (7)$$

Con esto finaliza la explicación sobre los sensores que hemos utilizado para el desarrollo de nuestra aplicación, a continuación seguiremos con los componentes encargados de la alimentación y comunicación del dispositivo.

12 APLICACIÓN ANDROID

Para realizar la implementación de la interfaz gráfica de usuario se ha decidido realizar una aplicación para dispositivos Android, debido a su reducido coste y la facilidad de mantenimiento. En los siguientes apartados, se realizará una breve introducción al sistema Android y se explicará cómo se ha realizado la implementación. En la Figura 19 se pueden ver los tipos de dispositivos en los que es posible instalar el sistema operativo Android.

Figura 19. Dispositivo con compatibles SO Android

12.1 ¿Qué es Android?

Android es un sistema operativo desarrollado inicialmente por la empresa Android Inc. y basado en Linux, un núcleo de sistema operativo libre, gratuito y multiplataforma. Fue un sistema operativo prácticamente desconocido hasta que Android Inc. fue comprada por Google.

Inicialmente, Android, fue diseñado únicamente para su uso en dispositivos móviles de pantalla táctil, es decir, Smartphone y Tablet. Sin embargo, en la actualidad su uso se

ha ampliado hasta los relojes y televisiones inteligentes, e incluso como sistema operativos para determinados coches.

Pese a que Android es una plataforma Open Source muchos de los dispositivos que actualmente se encuentran en el mercado no emplean únicamente código libre, sino que lo mezclan con software propio de cada compañía.

12.2 ¿Por qué usar Android?

Los motivos por los cuales un gran número de desarrolladores emplean Android como plataforma para la creación y distribución de sus aplicaciones son muy diversos. A continuación se enumeran algunos de los más destacados:

- Desde mediados del año 2010 y hasta la actualidad Android es el sistema operativo para móviles con mayor cuota de mercado, aumentando la diferencia con sus principales competidores año a año. En la siguiente figura se muestra la evolución del mercado.

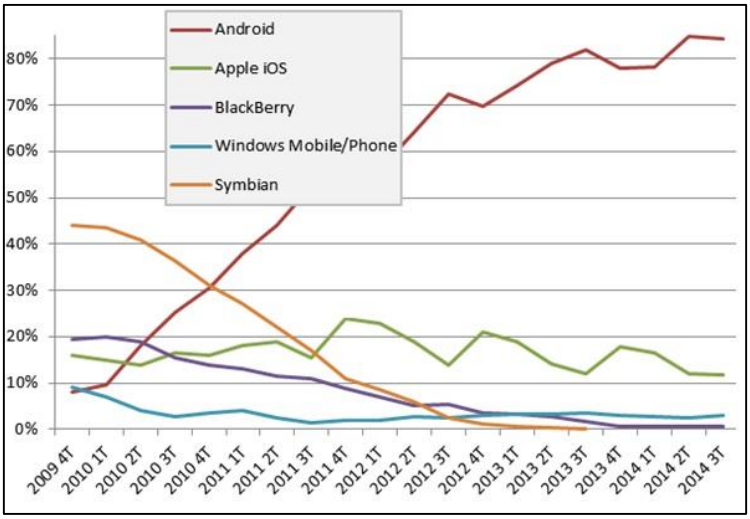

Figura 20. Evolución del mercado de SO móviles **[18]**

- Todo el software necesario para el desarrollo de aplicaciones Android es totalmente gratuito y, además, puede ser ejecutado en cualquier sistema operativo, ya sea Windows, Linux o Mac Os.

- Para poder desarrollar en Android únicamente es necesario pagar una sola cuota de 25$ para poder publicar tus aplicaciones en Google Play. Esto no sucede otros sistemas operativos. Por ejemplo, para poder desarrollar en el

sistema operativo de Apple, iOS, es necesario el pago de una cuota anual de 99$.

- En cuanto a la fase de desarrollo y test de la aplicación, los dispositivos Android tienen una gran ventaja ya que permiten la instalación de aplicaciones de cualquier procedencia. Esto supone una gran ayuda puesto que no es necesario subir tu aplicación a Google Play para poder instalarla y puedes hacerlo directamente desde tu PC.
- Gracias a que se trata de un sistema con carácter Open Source, existe una gran cantidad de documentación relacionada con el desarrollo de aplicaciones en Android.
- Android emplea como base para la programación el lenguaje Java, el cual es un lenguaje mucho más simple que el empleado por el sistema iOS, Objective-C.

12.3 Desarrollo en Android

A continuación se explicaran los puntos más importantes acerca del desarrollo de aplicaciones en Android. Concretamente, se explicaran algunos conceptos más específicos empleados en el desarrollo de esta aplicación **[17]**.

12.3.1 Android SDK y Eclipse

Para poder desarrollar en Android son necesarias determinadas herramientas y APIs, agrupadas con el nombre Android SDK (Android Software Development Kit), que deben ser instaladas en el ordenador. Las herramientas del SDK se encargan de compilar todo el código junto con los datos, imágenes, vídeos y demás archivos en un paquete de instalación denominado APK. Este paquete contiene todo lo necesario para instalar la aplicación en un dispositivo Android.

Aparte, para poder emplear el SDK es necesario tener instalado en el ordenador un entorno de desarrollo. Para la realización de este proyecto hemos empleado Eclipse puesto que ha sido empleado durante el periodo de aprendizaje. Sin embargo, este no es el entorno de desarrollo oficial para la plataforma Android, actualmente se emplea el entorno Android Studio.

12.3.2 Google Play

Google Play es la tienda de software desarrollada por Android para dispositivos Android. Como anteriormente hemos mencionado para poder publicar aplicaciones los desarrolladores deben pagar una única cuota de 25$ como registro.

Google Play proporciona a los desarrolladores de las aplicaciones distintas herramientas para la gestión de su aplicación, es decir, podrán gestionar las ventas, comprobar cuáles son las tendencias del mercado y controlar en que segmento del mercado está teniendo el mayor éxito.

Previamente a la publicación de la aplicación se pueden configurar distintas opciones y permisos. Además, el desarrollador podrá escoger a que segmento del mercado dirigirá la aplicación cambiando distintas opciones, por ejemplo: en qué países estará disponible la aplicación, a qué tipo de dispositivos se dará acceso a la aplicación, los requisitos mínimos que deberán tener eso dispositivos y, por último, a qué precio estará disponible la aplicación.

Sin embargo, las aplicaciones pueden ser instaladas sin necesidad de haberlas publicado en Google Play, siempre y cuando se disponga del archivo APK. Este método es muy útil, sobre todo en el proceso de desarrollo durante la depuración de la aplicación. También pueden obtenerse aplicaciones de otras tiendas virtuales como Amazon Appstore y SlideMe.

12.3.3 Componentes principales de una aplicación

Los componentes que a continuación se describirán componen los bloques principales sobre los que sostiene toda aplicación. Cada uno de ellos supone una entidad por sí misma y juega un papel diferente, la unión de cada una de las partes establece el comportamiento de la aplicación como conjunto.

Un aspecto del que únicamente dispone el diseño del sistema Android es la posibilidad de iniciar componentes de otras aplicaciones, siempre y cuando se dispongan de los permisos necesarios. De esta manera, si el usuario, por ejemplo, necesitara capturar alguna imagen por medio de la cámara, seguramente ya exista una aplicación en el dispositivo que permita realizar está acción. Con lo cual no será necesario escribir un

nuevo código para capturar la imagen, únicamente será necesario usar la aplicación ya desarrollada para este propósito.

Los componentes principales de cualquier aplicación son cuatro distintos, cada uno de ellos tiene un objetivo y un ciclo de vida distinto que define como se crea y destruye:

- ***Activity:*** proporciona la interfaz con la que interactúa el usuario.
- ***Service:*** se encarga de realizar las acciones en segundo plano y no proporciona una interfaz con la que interactuar.
- ***Content Provider:*** controla los datos compartidos entre aplicaciones.
- ***Broadcast Receiver:*** responde a los anuncios de acciones emitidos por el sistema.

Además de estos componentes principales, sobre los que se sostiene la aplicación diseñada, emplearemos también otro componente menos importante para cualquier aplicación, pero que en la nuestra adquirirá importancia. El componente es el siguiente:

- ***Adapter*****:** un *Adapter* es un objeto, el cual actúa como enlace entre una vista y los datos asociados a la misma.

A continuación procederemos a explicar con más detalle cada uno de los componentes mencionados.

12.3.3.1 Activity

Una *Activity* o actividad es el componente que proporciona al usuario la interfaz con la que interactuar a través de un *Layout*, que será explicado posteriormente. Cada aplicación suelen formarla distintas actividades relacionadas entre ellas. De todas estas *Activity* habitualmente una actúa como principal y suele denominarse "*Main Activity*". Esta actividad principal es la que se le presenta al usuario una vez abre la aplicación. Además, desde una actividad se puede acceder a nuevas actividades en la que se visualiza nuevo contenido. Cada vez que cambiamos de *Activity* la anterior queda almacenada en una pila llamada "back stack" que permitirá recuperarla en el futuro.

Las actividades poseen un ciclo de vida que es comunicado al sistema por medio de métodos de control del ciclo de vida. Existen métodos para las acciones de creación, destrucción, parada o reinicio de una actividad. Dentro de estos métodos se puede programar código que se ejecutará cada vez que se realice el cambio de estado que controle ese método.

Los principales métodos para el control del ciclo de vida de una *Activity*, mostrado en la Figura 21, son los siguientes **[19]**:

- **onCreate():** este método es ejecutado la primera vez que se inicia la *Activity*. En él se deben inicializar todas las componentes principales de la actividad, además de usar el método setContentView() para establecer el layout de la interfaz de dicha *Activity*. Una vez se ha ejecutado este método seguidamente se ejecuta OnStart().
- **onStart():** método ejecutado inmediatamente antes de mostrar la actividad al usuario.
- **onResume():** se ejecuta cuando la *Activity* está lista para interactuar con el usuario de la aplicación. En ese momento esa actividad se encuentra en la parte más alta de la pila "back stack".
- **onRestart():** ejecutado cuando una actividad en estado "detenido" vuelve a mostrarse de nuevo al usuario.
- **onPause():** este método se ejecuta a la primera indicación de que se abandona la *Activity*. Deberemos usar este método para guardar los datos necesarios para la siguiente ejecución de la actividad y detener los elementos de la aplicación que consuman mucha CPU. Todo el código de este método debe ser de rápida ejecución puesto que no permitirá el inicio de la siguiente *Activity* hasta que se haya completado.
- **onStop():** este método es ejecutado cuando la *Activity* no va a volver a ser utilizada por el usuario, es decir, no va a volver a ser visualizada.
- **onDestroy():** método ejecutado antes de que la *Activity* sea destruida.

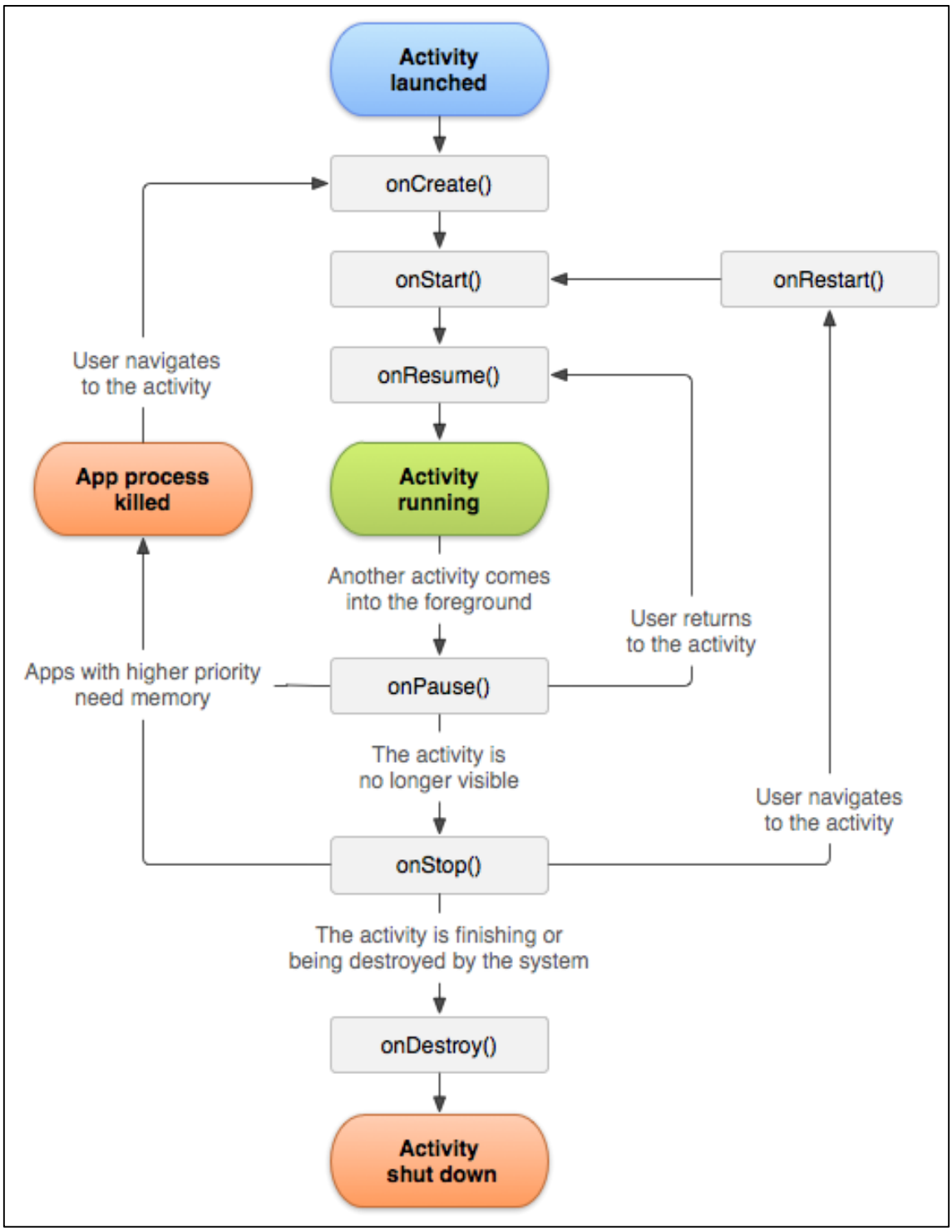

Figura 21. Ciclo de vida de una Activity **[19]**

12.3.3.2 Service

El *Service* es un componente que no proporciona al usuario una interfaz en el dispositivo, sino que únicamente ejecuta acciones en segundo plano. Un *Service* puede ser ejecuta desde cualquier otro componente y continuar ejecutándose en segundo plano incluso si se sale fuera de la aplicación. Por otra parte, se puede vincular

cualquier componente con un *Service* y de esta manera interactuar e intercambiar información con él.

Un *Service* puede ser de dos tipos distintos:

- ***Started Service*:** en este caso el *Service* es creado por medio del método startService(). Cuando el *Service* ha sido iniciado este permanece ejecutándose en segundo plano indefinidamente hasta que el método stopService() es utilizado. Un *Started Service* queda automáticamente destruido por el sistema cuando es detenido.

- ***Bound Service*:** este otro tipo de *Service* se crea empleando el método bindService(). Esto ofrece al componente que actúa como cliente una conexión con el *Service* por medio de la cual podrán interactuar. Además, de esta manera, podremos vincular varios componentes con un solo *Bound Service.* El *Service* se destruye automáticamente cuando todos los componentes se han desvinculado.

Pese a ser distintos, los dos tipos de *Service* no están del todo separados. Un *Started Service* puede perfectamente estar vinculado con otros componentes por medio del método onBind(). Sin embargo, éste únicamente será destruido cuando el método stopService() sea llamado.

Cuando empleamos un *Service* debemos tener cuidado con la cantidad de código desarrollado en él, debido a que no se crea un nuevo hilo para su ejecución, sino que se ejecuta en el hilo principal de la aplicación. Por lo tanto, si realizamos acciones que consuman gran cantidad de recursos de la CPU puede producirse el bloqueo de la aplicación.

Del mismo modo que en una *Activity*, un *Service* también tiene un ciclo de vida, aunque mucho más simple que el de las actividades, que es controlados por métodos llamados cuando se producen un cambio en su estado. Debido a que su ejecución se produce en un segundo plano, es más complicado saber en qué estado se encuentra en cada momento, por lo tanto, deberemos prestar mayor atención a los métodos de

control del ciclo de vida, que puede verse en la Figura 22. Dichos métodos son los siguientes **[20]**:

- **onBind():** con este método se inicia el vínculo entre componentes a través del cual se producirá el intercambio de información.
- **onUnbind():** este método es ejecutado cuando los componentes clientes del *Service* han sido desvinculados.
- **onStartComand():** se ejecuta cuando un componente inicia un *Started Service.*

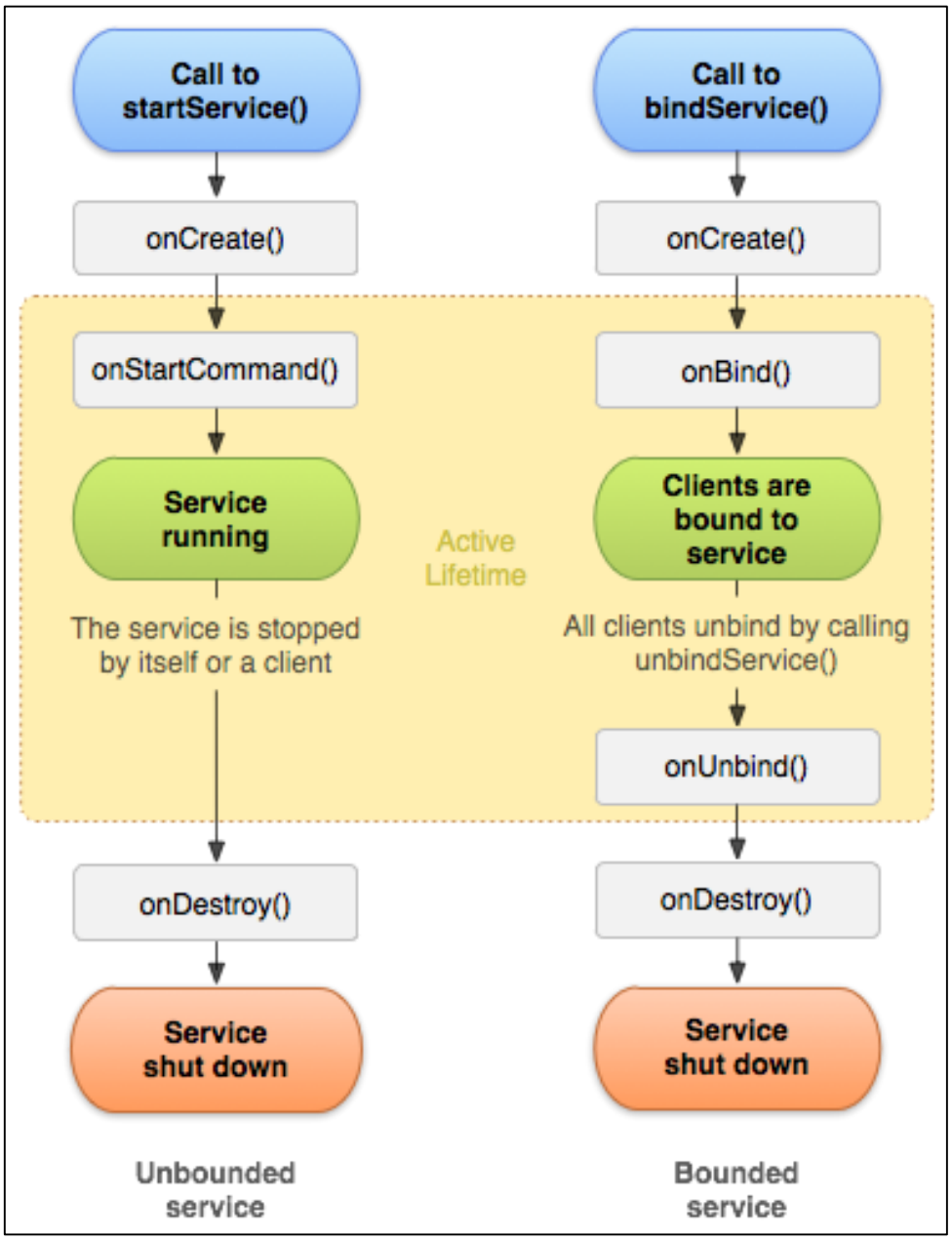

Figura 22. Ciclo de vida de un Service **[20]**

12.3.3.3 Content Provider

Este componente es el encargado de gestionar los datos compartidos entre distintas aplicaciones. Permite guardar los datos en distintas localizaciones a las que pueda tener acceso la aplicación como puede ser una base de datos, un archivo de sistema o la web.

Por medio de un *Content Provider* otras aplicaciones podrán buscar o incluso modificar datos en tu aplicación. Por ejemplo, los datos del usuario son controlados por un *Content Provider* y, a través de este, las aplicaciones con permisos suficientes serán capaces de acceder a estos datos y modificarlos.

12.3.3.4 Broadcast Receiver

Un *Boadcast Receiver* es el encargado de responder a los anuncios de acciones emitidos por el sistema. Los anuncios emitidos son muy numerosos, por ejemplo se pueden citar, el nivel de la batería o si se ha apagado la pantalla. En nuestro caso, emplearemos sobre todo los anuncios relacionados con las acciones del Bluetooth. Se debe tener en cuenta que el sistema no es el único que puede realizar este tipo de anuncios, ya que las aplicaciones también pueden realizarlos, por ejemplo, para indicar la finalización de una descarga.

Un *Broadcast Receiver* carece de interfaz gráfica, por lo tanto, debe crearse una barra de notificación para indicar cuando se produce un evento de este tipo. Estos componentes tienen la función de avisar o de ejecutar otros componentes cuando un evento de estos ocurre.

12.3.3.5 Adapter

Un *Adapter* es un objeto, el cual actúa como enlace entre una vista y los datos asociados a la misma. Además, proporciona acceso a los elementos que contiene datos y proporciona una vista a cada uno de esos elementos para el conjunto final de los datos.

Existen diversos tipos de *Adapter* según el tipo de vista para el que lo vayamos a emplear, es decir, una lista, una lista desplegable, etc. En nuestro caso vamos a

emplearlo para realizar una lista de los dispositivos *SensorTag* disponibles, por lo tanto, el *Adapter* que emplearemos en nuestra aplicación es *BaseAdapter.*

12.3.4 Android Manifest

Dentro de una aplicación Android existe un archivo denominado *AndroidManifest.xml*. En él se deben declarar todos los componentes que vayan a ser utilizados por dicha aplicación puesto que el sistema debe tener constancia de que existen los componentes que van a ser utilizados.

- Aparte de la declaración de los componentes, el Manifest tiene las siguientes funciones:
- Identificar los permisos necesarios para la aplicación, en nuestro caso será el del acceso al Bluetooth.
- Establecer cuál será la versión mínima de Android que deberá tener el dispositivo en el que sea instalada la aplicación. En nuestro caso será la versión de Android 4.3.
- Declarar los requerimientos de hardware o software que necesitará la aplicación para un correcto funcionamiento. Por ejemplo, en el caso de este proyecto, requerirá el uso del Bluetooth 4.0 o Low Energy.
- Añadir las librerías API que necesiten estar vinculadas a la aplicación. Por ejemplo, la API de Google Maps.

12.3.5 Layout

Un *Layout* constituye la estructura visual de la interfaz mostrada en la pantalla por medio de la *Activity.* El *Layout* puede ser creado de dos maneras distintas:

- **Por medio de un archivo XML** en el que empleando un vocabulario específico proporcionado por Android para poder declarar y modificar los distintos elementos del *Layout.*

- **Instanciar los elementos en tiempo de ejecución** por medio de código que programará la creación de vistas o subclases de las mismas mientras la aplicación se está ejecutando.

De esto dos métodos es preferible emplear el primero de ellos puesto que permite una mejor separación entre la interfaz gráfica y el código que controla su comportamiento. Realizando la descripción de la interfaz de esta manera no es necesario modificar o recompilar el código de la aplicación puesto que las descripciones son externas al código de la aplicación.

Existe variedad de *Layouts* que se adecúan a las distintas necesidades de presentación de la información en la aplicación. Son los siguientes:

- ***Grid Layout* (Figura 23):** los componentes en este *Layout* se organizan en un mallado de filas y columnas.

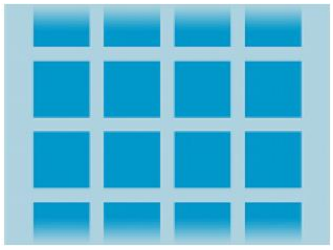

Figura 23. Grid Layout.

- ***Linear Layout* (**Figura **24):** en este caso los componentes se organizan en filas o columnas dependiendo de si el *Layout* es horizontal o vertical.

Figura 24. Linear Layout

- ***Relative Layout* (Figura 25):** no permite especificar exactamente dónde queremos colocar cada uno de los componentes de la interfaz sin restricción alguna.

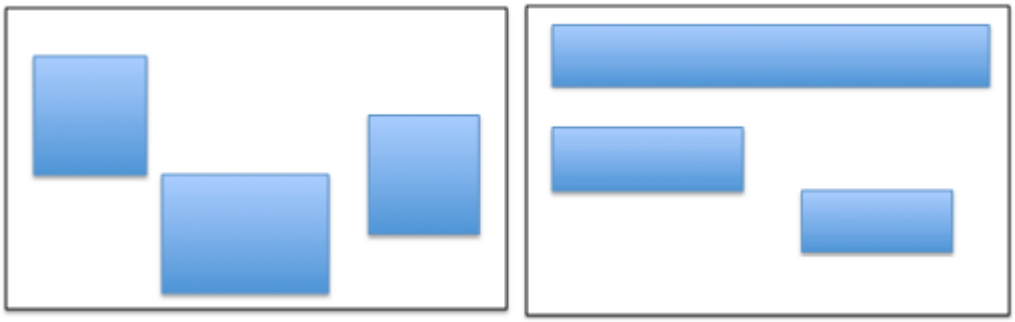

Figura 25. Relative Layout

- ***Frame Layout* (Figura 26):** este *Layout* permite incluir distintas interfaces dentro de una misma *Activity*. Podremos pasar de una interfaz a otra mediante acciones como puede ser la pulsación de un botón. De este modo se distribuyen más ordenadamente los componentes.

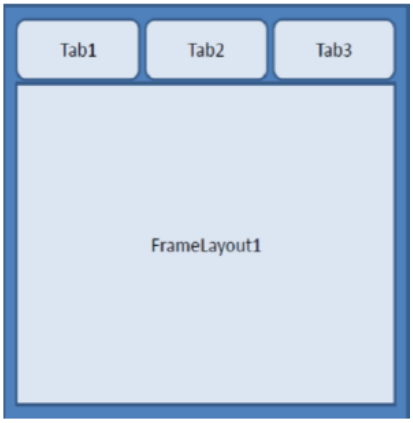

Figura 26. Frame Layout

- ***Table Layout* (Figura 27):** este *Layout* está compuesto por un número de objetos denominados *TableRow* que constituye una fila de la tabla. A su vez cada *TableRow* puede estar dividida en varias celdas que contendrán los componentes. El número de columnas de este *Layout* lo define la fila con mayor número de celdas.

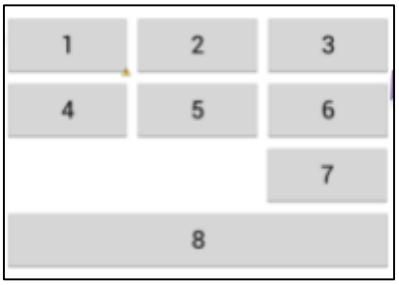

Figura 27. Table Layout

12.3.6 Bluetooth

A lo largo de este apartado desarrollaremos las clases clave que necesitaremos emplear para poder realizar una conexión Bluetooth clásica con cualquier dispositivo.

Las clases necesarias para poder realizar la conexión son las siguientes **[21]**:

- ***BluetoothAdapter*:** esta clase representa la antena de Bluetooth del dispositivo móvil en el cual está instalada la aplicación. Se trata del punto por donde se realiza la entrada a la interacción con la información procedente del Bluetooth. Con este objeto se pueden realizar las acciones necesarias para realizar una conexión, es decir, buscar otros dispositivos disponibles, obtener la lista de los dispositivos emparejados con el nuestro, condición necesaria para establecer una conexión, crear un objeto de la clase *BluetoothDevice* a partir de la dirección MAC del dispositivo externo y crear un objeto *BluetoothSocket* para gestionar la conexión.

- ***BluetoothDevice*:** representa a un dispositivo Bluetooth remoto. Esta clase es empleada para crear un conexión utilizando un *BluetoothSocket*, además de para pedir distintas informaciones al otro dispositivo, como pueden ser el nombre, la dirección, el estado, etc.

- ***BluetoothSocket*:** clase empleada para representar el hilo a través del cual se produce la comunicación entre los dos dispositivos. Permite realizar el intercambio de información a través de otras dos clases llamadas *ImputStream*, para comunicaciones entrantes, y *OutputStream*, para comunicaciones salientes.

- ***ImputStream*:** esta clase se utiliza para realizar la recepción de los datos a través del Bluetooth. Para recibir los datos emplea el método *read().*

- ***OutputStream*:** usada para realizar el envío de los datos a través del canal de comunicación. Para realizarlo emplea el método *write().*

12.3.7 Bluetooth Low Energy

El Bluetooth 4.0, también denominado Bluetooth Low Energy, es el último avance desarrollado para la tecnología Bluetooth y fue introducido en Android a partir de la versión 4.3. Esta tecnología ha sido desarrollada para poder realizar la comunicación reduciendo significativamente el consumo de energía del dispositivo. De esta manera, esta reducción de energía permite la comunicación de dispositivos Android con dispositivos que dispongan de Bluetooth Low Energy y requieran muy poca energía para su funcionamiento, como pueden ser pulsímetros, dispositivos deportivos o, como en nuestro caso, sensores.

12.3.7.1 Conceptos y clases básicos

Para poder implementar el código necesario para realizar la conexión por medio del Bluetooth Low Energy es necesario utilizar diversas clases específicas para este cometido. Las clases empleadas son las siguientes **[22]**:

- ***General Attribute Profile (GATT)*:** el perfil GATT es un perfil general para enviar y recibir pequeños paquetes de datos denominados *attributes* (atributos) por medio de una conexión de Bluetooth Low Energy. Todos los perfiles de una aplicación Low Energy están basados en GATT.

- ***Bluetooth Characterisitic* (Característica):** contiene un valor y diversos descriptores que definen el valor de la característica. Una característica puede ser considerada como un tipo, similar a una clase.

- ***Bluetooth Descriptor* (Descriptor):** un descriptor es un atributo definido que describen el valor de una característica. Por ejemplo, un descriptor podría describir el rango aceptable o la unidad de medida del valor de una característica.

- ***Bluetooth Service* (Servicio):** por último, un servicio es un grupo de diversas características. Por ejemplo, en el caso que no ocupa, un servicio sería el sensor de temperatura y sus características el valor que proporciona y el valor para activarlo.

Una vez definidos los conceptos básicos sobre la tecnología Low Energy, para su manejo es necesario el uso de las clases descritas a continuación:

- ***BluetoothGatt*:** esta clase proporciona la funcionalidad necesaria para poder establecer la conexión con un dispositivo.
- ***BluetoothGattCallback*:** es necesario crear esta clase para poder establecer una conexión con un dispositivo remoto. Una vez hemos creado la clase emplearemos el método *connectGatt()* para obtener un objeto de la clase *BluetoothGatt.*
- ***BluetoothGattCharacteristic*:** empleada para representar cada una de las características del dispositivo remoto.
- ***BluetoothGattService*:** representa cada uno de los *Service* del dispositivo.

Una vez explicados los conceptos básicos procederemos a desarrollar el proceso llevado a cabo para crear la aplicación. Para ello, en los apartados que siguen, explicaremos el diseño y la implementación llevados a cabo.

12.4 Diseño e implementación de la aplicación

Tras haber explicado los conceptos básicos sobre el desarrollo de aplicaciones en Android y sobre el Bluetooth Low Energy procederemos con la explicación del procedimiento seguido para el desarrollo de la aplicación.

12.4.1 Diseño de la interfaz

En este apartado realizaremos la descripción de la estructura y el funcionamiento de la aplicación. La aplicación se trata de un sistema de monitorización de los datos proporcionados por el sensor a través del Bluetooth.

La aplicación está estructurada en diferentes actividades a las que se irá accediendo según se profundice en el uso de la aplicación, la primera *Activity* que abrirá la aplicación será, como suele ser habitual, la *MainActivity*. Las actividades son las siguientes:

- ***Main Activity*:** en esta *Activity* se dará acceso a las diferentes opciones de la aplicación.

- ***SensorTag Search Activity*:** *Activity* que permitirá al usuario buscar los dispositivos *SensorTag* disponibles para recibir los datos de sus sensores. Una vez el usuario ha seleccionado un dispositivo se procederá a establecer la conexión con el mismo.

- ***Sensor Read Activity*:** esta *Activity* se mostrarán los sensores disponibles del dispositivo y el usuario de la aplicación podrá escoger de cuál de ellos desea obtener datos. Además al iniciar esta actividad iniciaremos también un *Service* que se encargará de mantener la conexión entre el dispositivo móvil y el *SensorTag*.

- ***Device Selection Activity*:** emplearemos esta *Activity* para buscar el dispositivo de control a través del cual controlaremos los parámetros del sistema domótico y los ajustaremos según nuestras necesidades y los datos recibidos, a través de nuestra aplicación, de los sensores de humedad y temperatura. Una vez hayamos seleccionado un dispositivo se iniciará el *Service* encargado de gestionar la conexión Bluetooth entre ambos dispositivos.

- ***Change Values Activity*:** por último, a través de esta *Activity* estableceremos los valores de configuración del sistema domótico y que serán enviados el dispositivo de control al cual estamos conectados.

A parte de las actividades ya mencionadas, también se han implementado dos *Services* diferentes para gestionar las conexiones Bluetooth, uno para la conexión con el *SensorTag*, ya que se realiza a través de la tecnología Low Energy, y otro para la conexión con el dispositivo de control domótico, puesto que este caso la conexión emplea Bluetooth normal:

- ***Bluetooth Service*:** este *Service* es el encargado de manejar la conexión con el *SensorTag* por medio de la tecnología Low Energy.

- ***Connect Service*:** encargado de gestionar la conexión, por medio de Bluetooth, con el dispositivo de control domótico.

Estos *Service* contienen los métodos necesarios para poder realizar la comunicación entre los dos dispositivos conectados. El motivo por el cual se ha decidido emplear un *Service* para cada una de estas conexiones es la necesidad de poder acceder a la conexión desde cualquier *Activity*, ya que si realizásemos la conexión empleando una actividad la conexión se cerraría al navegar entre actividades.

De los dos tipos de *Service* explicados anteriormente en este documento, se han empleado ambos tipos. En el encargado de gestionar la conexión Low Energy se ha decidido utilizar un *Bound Service* ya que el *Service* se tiene que destruir cuando se desvincula de su *Activity*. Sin embargo, para el otro, se ha implementado una combinación de ambos tipos, puesto que al tener que vincularse a varias actividades distintas es necesario el uso del método onBind(). Además, necesitamos controlar la creación y la destrucción del *Service* tendremos que utilizar los métodos startService() y stopService().

Además de las *Activity* y *Service* ya mencionados, encargados de la gestión principal de la aplicación, también se han implementado diversas clases de menor importancia, pero totalmente necesarias para el correcto funcionamiento de la aplicación. Por un lado se han creado dos *Adapter*, que son los siguientes:

- ***DeviceList Adapter*:** *Adapter* empleado para crear la vista en la cual se mostrará la lista de *SensorTag* disponibles para conectar con el nombre y la dirección de cada uno.
- ***SensorList Adapter*:** en este caso se visualizará la lista de sensores disponibles para mostrar sus datos, cada uno con su dirección UUID.

El resto de clases se han creado para la gestión de los datos de los sensores de una manera más eficiente. Son las siguientes:

- ***SensorTag*:** en esta clase se almacenan el nombre y la dirección de los sensores que contiene el *SensorTag*.
- ***Sensor*:** clase abstracta en la que se incluyen los métodos necesarios para uso de los sensores.

- ***Temperature Sensor*:** esta clase implementará los métodos particulares para el uso del sensor de temperatura.
- ***Humidity Sensor*:** esta clase implementará los métodos particulares para el uso del sensor de humedad.

A continuación, en el siguiente apartado, se llevará a cabo la explicación de la implementación que se ha realizado de cada una de las partes de la aplicación explicada en este apartado.

12.4.2 Implementación de la interfaz

Tras haber realizado la explicación del diseño y la estructura que tendrá la aplicación, a continuación procederemos a desarrollar cuál ha sido la implementación llevada a cabo en cada *Activity*, *Service* y demás componentes principales de la aplicación. En el apartado ANEXOS se podrán ver los diagramas de flujo de las distintas actividades, además del ciclo de vida de la aplicación completa.

12.4.2.1 Aspectos generales

A continuación se explicará algunos aspectos generales de la aplicación desarrollada para este proyecto.

En primer lugar, el bloque principal de la aplicación se ha dividido en dos secciones diferenciadas con un nexo común. Estas dos secciones son, por un lado, la encargada de la conexión con los sensores y la recepción de los datos tomados por estos. Por el otro lado, tendremos la parte encargada de la conexión con el dispositivo controlador y el envío de los nuevos datos de los ajustes. Cada una de estas secciones contará con dos *Activity*, con las que interactuará el usuario y un *Service* que gestionará la conexión por Bluetooth. A ambas partes de la aplicación se accederá desde la *Main Activity*.

La explicación de la implementación realizada para cada *Activity* y *Service* se realizará por secciones, comenzando por la *Main Activity*. Después pasaremos a la sección de los sensores, es decir, *SensorTag Search Activity*, *Sensor Read Activity* y *Connect Service*. Y, por último, se explicará la sección del controlador, que son, *Device Selection Activity*, *Change Values Activity* y *Bluetooth Service*.

Una vez explicado el bloque principal, se procederá con la explicación del resto de clases que completan el código de la aplicación

Cabe destacar también, que para que no se produzca una deformación del diseño de la interfaz, todas las medidas de los componentes visuales se han implementado en densidad de píxeles para que las proporciones se mantengan cuando el tamaño de la pantalla del dispositivo en el que se instale cambien. Además, todas las capturas de la interfaz mostradas a lo largo de este documento han sido tomadas en el Smartphone utilizado para el desarrollo principal de la aplicación, un Huawei Ascend P7.

Una vez explicados estos aspectos generales de la aplicación se procederá con la explicación de la implementación realizada.

12.4.2.2 Main Activity

En esta *Activity* se muestra el menú principal, en el cual podremos escoger entre las distintas opciones que ofrece nuestra aplicación y acceder a las actividades de selección de dispositivos. La Figura 28 muestra una captura de pantalla de la actividad.

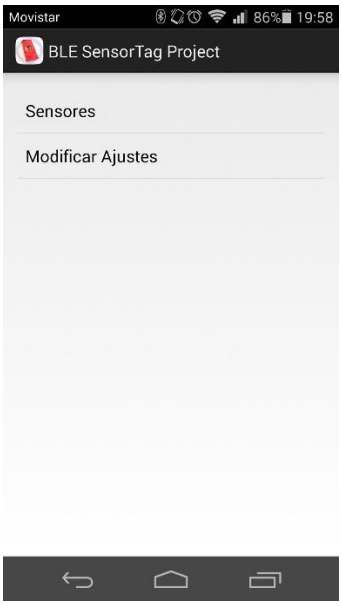

Figura 28. Captura de pantalla de Main Activity

La implementación de la *Main Activity* se ha realizado como una extensión de *ListActivity*, que consiste en un modelo de actividad estructurada para mostrar una serie de elementos en forma de lista. Los elementos contenidos en la lista están contenidos en una constante de tipo *String*, el cual se introduce en la lista empleando

el método propio de la *ListActivity setListAdapter()*. Los elementos de dicha lista son seleccionables y crean un enlace a una nueva actividad. Para iniciar la nueva actividad es necesario emplear un objeto denominado *Intent*. Este objeto nos permite enlazar, en tiempo de ejecución, dos componentes distintos de la aplicación. Además, representa la intención del usuario de realizar una acción, en este caso la acción consistiría en la apertura de una nueva actividad. Para poder emplear un *Intent* para abrir una nueva actividad es necesario proporcionarle dos datos distintos:

- Por un lado deberemos pasarle el entorno de la aplicación, denominado *Context* o contexto.
- Por otro, tendremos que indicar el nombre de la actividad que queremos abrir.

Una vez le hemos proporcionado ambos datos en la creación del *Intent*, usaremos el método *startActivity()*, que recibe como parámetro el *Intent*, para iniciar la nueva actividad, a continuación, en la Figura 29, se muestra el código necesario para abrir la nueva actividad:

```
@Override
protected void onListItemClick(ListView l, View v, int position, long id) {

    Intent myIntent = null;

    switch(position){
    case 0:
        myIntent = new Intent(v.getContext(), SensorTagSearchActivity.class);
        startActivity(myIntent);
        break;
    case 1:
        myIntent = new Intent(v.getContext(), DeviceSelectionActivity.class);
        startActivity(myIntent);
        break;
    }
}
```

Figura 29. Código para iniciar nueva Activity

Seguidamente procederemos con la explicación de la actividad que nos permitirá seleccionar el *SensorTag* sobre el cual deseamos leer los datos.

12.4.2.3 SensorTag Search Activity

Esta *Activity* se encarga de mostrar la lista de los dispositivos *SensorTag* disponibles para establecer conexión. Una vez seleccionemos el dispositivo con el cual deseamos

conectar accederemos a la actividad que nos permitirá ver los datos capturados por los sensores. En la figura ***x*** se muestra el diseño de la actividad.

En este caso, al igual que sucedía con la *Main Activity*, esta actividad se extiende de una *List Activity*. A diferencia de la anterior *Activity*, en este caso, la lista de dispositivos no está almacenada en un *String*, sino que dependerá de los dispositivos encontrados al realizar la búsqueda. Para comenzar la búsqueda de dispositivos habrá que pulsar el botón *Scan*, y en el caso de que deseemos detenerla pulsaremos el botón *Stop*. Si no detenemos manualmente la búsqueda esta se detiene automáticamente tras 10 segundos.

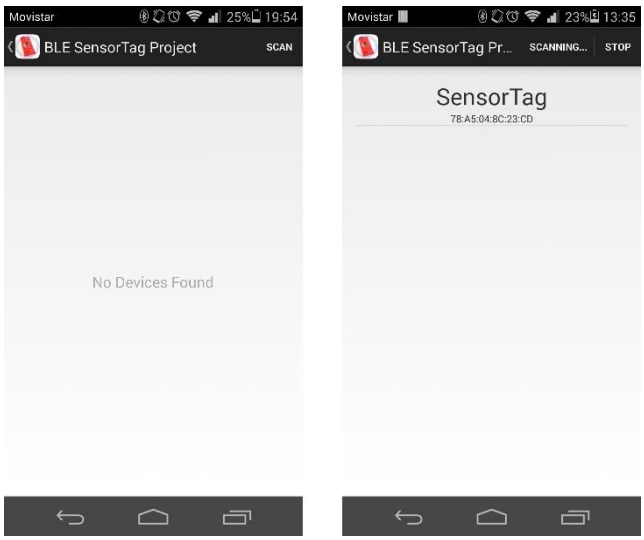

Figura 30. Capturas de pantalla de SensorTag Search Activity

Cuando pulsamos el botón *Scan*/*Stop* se produce una llamada al método *startScanLeDevice()*, que recibe como parámetro *true* o *false* según queramos activar o detener la búsqueda. Cuando el botón pulsado es *Scan* introducimos *true* para iniciar la búsqueda, en el caso contrario, el parámetro introducido es *false* para detenerla. La Figura 32 muestra el código necesario para iniciar la búsqueda.

Para almacenar los dispositivos encontrados en la búsqueda utilizamos un objeto de la clase *DeviceList Adapter* al que hemos denominado *DeviceList*, empleando el método *addDevice()* (Figura 31). Para mostrar los dispositivos, al igual que en *Main Activity*, empleamos el método *setListAdapter()*, pero en este caso recibirá como parámetro el *Device List*.

```
if (device.getName().equals("SensorTag")) {
    DeviceList.addDevice(device);
}
```

Figura 31. Código para añadir un dispositivo a la lista

```
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.device_scan, menu);
    if(!isScanning){
        menu.findItem(R.id.scanning).setVisible(false);
        menu.findItem(R.id.start_scan).setVisible(true);
        menu.findItem(R.id.stop_scan).setVisible(false);
    }else{
        menu.findItem(R.id.scanning).setVisible(true);
        menu.findItem(R.id.start_scan).setVisible(false);
        menu.findItem(R.id.stop_scan).setVisible(true);
    }
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();
    if (id == R.id.start_scan) {
        startScanLeDevice(true);
        invalidateOptionsMenu();
        return true;
    }
    if (id == R.id.stop_scan) {
        startScanLeDevice(false);
        invalidateOptionsMenu();
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

Figura 32. Código para iniciar o detener búsqueda

Una vez hemos decidido cuál de los *SensorTag* disponibles queremos monitorizar tendremos que seleccionarlo. Una vez lo hayamos seleccionado emplearemos un objeto *Intent* para iniciar una nueva *Activity* llamada *SensorRead*. En este caso dentro del *Intent*, antes de usarlo para iniciar la nueva actividad, incluiremos los datos necesarios del dispositivo, esto es, el nombre y la dirección del mismo, para ello utilizaremos el método propio de la clase *Intent putExtra()*. Así mismo, emplearemos estos datos para iniciar la conexión a través de un *Service*. En la Figura 33 se muestra el código explicado.

En el siguiente apartado desarrollaremos la explicación sobre la actividad *Sensor Read Activity* que emplearemos para leer los datos recibidos de los sensores.

```
public void onListItemClick(ListView parent, View v, int position, long id) {
    Log.i(TAG,"ItemSelected");
    BluetoothDevice device = DeviceList.getDevice(position);
    Intent myIntent = null;
    myIntent = new Intent(v.getContext(), SensorReadActivity.class);
    myIntent.putExtra(SensorReadActivity.EXTRAS_DEVICE_NAME,
            device.getName());
    myIntent.putExtra(SensorReadActivity.EXTRAS_DEVICE_ADDRESS,
            device.getAddress());
    startActivity(myIntent);

}
```

Figura 33. Código para incluir nombre y dirección en el Intent

12.4.2.4 Sensor Read Activity

Utilizaremos esta *Activity* para realizar la monitorización de los datos procedentes de los sensores que incorpora el dispositivo *SensorTag*, concretamente los datos de los sensores de humedad y temperatura. Podremos seleccionar de cuál de los sensores deseamos mostrar los datos. La siguiente figura el diseño realizado para esta actividad.

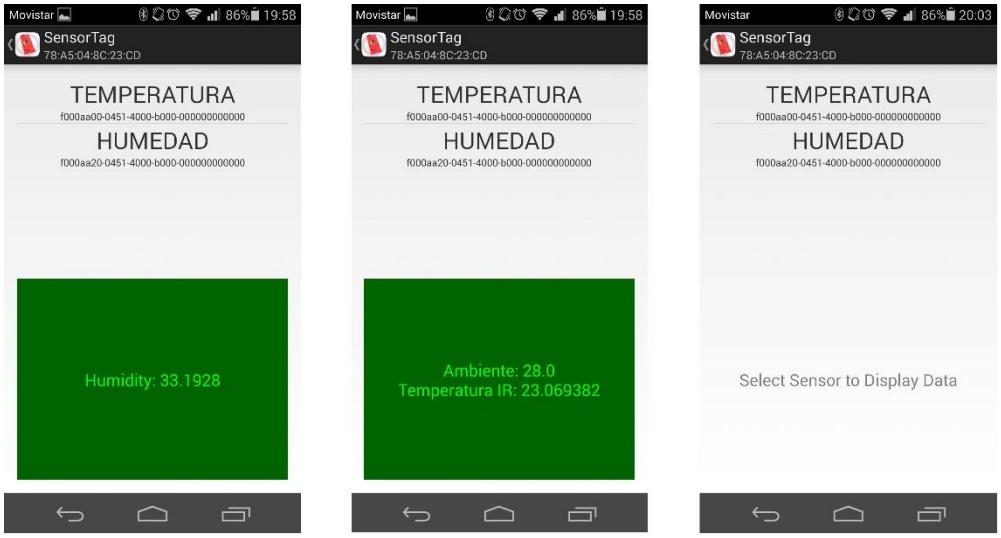

Figura 34. Capturas de pantalla de Sensor Read Activity

Esta actividad estará asociada al *Service* llamado *Connect Service*, está asociación ser realizará por medio del método *bindService()*, con lo cual estaremos creando un *Bound Service*. Cuando la asociación se produzca además iniciaremos la conexión con el dispositivo empleando el método del *Connect Service connect()*. Esta conexión se cerrará al mismo tiempo que se finalizará la asociación con el *Service*, empleando los

métodos *unbindService()* y *disconnect()*. De la Figura 35 a la Figura 37 se muestra el código implementado.

```
final Intent connectionService = new Intent(this, ConnectService.class);
bindService(connectionService, ServiceConnection, BIND_AUTO_CREATE);
```

Figura 35. Código para asociar a un Service

```
@Override
protected void onResume() {
    super.onResume();
    registerReceiver(receiver, receiverIntentFilter());
    if (mConnectService != null) {
        final boolean result = mConnectService.connect(DeviceAddress);
        Log.i(TAG, "Connect request result=" + result);
    } else
        Log.w(TAG, "ConnectService Not Started");
}
```

Figura 36. Código para conectar con el dispositivo

```
@Override
protected void onDestroy() {
    super.onDestroy();
    mConnectService.disconnect();
    unbindService(ServiceConnection);
    mConnectService = null;
}
```

Figura 37. Código para desconectar el dispositivo

Así mismo, debido a que el *Service* se encarga de gestionar una conexión con un dispositivo Low Energy, emite diversos anuncios a los que debemos responder. Para responder a estos anuncios, como hemos explicado anteriormente, hay que implementa un *Broadcast Receiver* (Figura 38. Código del *Broadcast Receiver*. Los anuncios se emitirán cuando el *ConnectService* cambie el estado de la conexión con el dispositivo remoto, cuando un servicio (*BluetoothGattService*) del dispositivo sea descubierto, en nuestro caso corresponderá con los sensores disponibles, y, por último, cuando se produzca alguna acción en las características (*BluetoothGattCharacteristic*) del sensor seleccionado, ya sea lectura, escritura, o cambio del valor almacenado en la misma. La emisión de anuncios se abordará con más detalle en el apartado en el que explicaremos el *Connect Service*.

```
private final BroadcastReceiver receiver = new BroadcastReceiver() {

    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (ConnectService.ACTION_GATT_CONNECTED.equals(action)) {
            Log.i(TAG, "Action gatt connected");
            isConnected = true;
            BLeGatt = mConnectService.getBluetoothGatt();
            invalidateOptionsMenu();
        } else if (ConnectService.ACTION_GATT_DISCONNECTED.equals(action)) {
            Log.i(TAG, "Action gatt disconnected");
            isConnected = false;
        } else if (ConnectService.ACTION_GATT_SERVICES_DISCOVERED
                .equals(action)) {
            Log.i(TAG, "Action gatt discovered");
            addServices();
        } else if (ConnectService.ACTION_DATA_AVAILABLE.equals(action)) {
            displayData(intent);
        }
    }

};
```

Figura 38. Código del Broadcast Receiver

En cuanto al diseño visual de la actividad, se ha decido dividir la pantalla en dos secciones diferenciadas:

- En la primera de ellas se mostrará una lista de los sensores disponibles para mostrar sus datos.
- En la segunda podremos ver los datos del sensor seleccionado.

Para realizar la lista de sensores emplearemos un objeto de la clase *Sensors List Adapter* que hemos denominado *sensorsList*. En este objeto añadiremos lo sensores disponibles una vez hayan sido encontrados por el *Connect Service*. Para ello, cuando el *Broadcast Receiver* reciba el anuncio que indica que los servicios del *SensorTag* han sido descubiertos, se llamará al método *addServices()* (Figura 39. Código para añadir un servicio (sensor) a la lista. Una vez han sido añadidos los sensores, el usuario tendrá la posibilidad de escoger de cuál de ellos desea mostrar información. Una vez ha escogido un sensor y lo ha pulsado, los datos de este se mostrarán en la segunda sección de la actividad.

```
private void addServices() {
    services = mConnectService.getServices();
    sensorsList = new SensorsListAdapter(getBaseContext(), services);
    sList.setAdapter(sensorsList);
}
```

Figura 39. Código para añadir un servicio (sensor) a la lista

La visualización de los datos también se producirá de manera síncrona con la recepción del anuncio que indica que hay nuevos datos en el sensor par ser leídos. Este anuncio se enviará por primera vez cuando seleccionemos el sensor y se lean los datos del mismo. A partir de entonces, se volverá a enviar cada vez que se produzca un cambio en los datos. Tras haber recibido el anuncio se llamará al método encargado de actualizar la sección de visualización de los datos *displayData()*, el código se muestra en la Figura 40. Código para mostrar los datos del sensor.

```
private void displayData(Intent intent) {
    final String data = intent.getStringExtra(ConnectService.EXTRA_DATA);
    dataField.setText(data);
}
```

Figura 40. Código para mostrar los datos del sensor

Adicionalmente a la muestra de los datos, el diseño de esa sección de la actividad cambiará en función del valor que tengan los datos en ese momento concreto, teniendo en cuenta que los datos son tomados dentro de casa. Este cambio se producirá siguiendo el siguiente criterio:

- Para el sensor de temperatura emplearemos el color verde para cuando la temperatura se encuentre en valores adecuados, que será entre 20ºC y 30ºC, rojo para cuando el valor supere los 30ºC, indicando que la temperatura es demasiado elevada y azul para cuando no llegue a 20ºC, indicando que es demasiado baja.
- En el caso del sensor de humedad únicamente emplearemos el color verde y rojo. El primero de ellos para cuando el valor de humedad sea adecuado, es decir, entre 30% y 60% y rojo para el caso contrario y se encuentre fuera de ese rango.

A continuación explicaremos el *Service* encargado de gestionar la conexión con el dispositivo *SensorTag.*

12.4.2.5 Connect Service

A lo largo de este apartado procederemos con el desarrollo del *Service* encargado de establecer y gestionar la conexión de Bluetooth Low Energy. Como ya hemos explicado anteriormente, al tratarse de un *Service* carece de interfaz gráfica o *Layout*, en consecuencia el usuario no podrá interactuar directamente con el mismo.

En primer lugar se han creado los distintos objetos y métodos necesarios para poder vincular el *Service* a la *Activity*. Concretamente se ha creado un objeto llamado *IBinder* que es el que permitirá que se produzca la vinculación. Además, también ha sido necesario implementar un método que hará que la *Activity* a la que se desea vincular el *Service*, este método se denomina *onBind()* que devolverá el objeto *IBinder* la primera vez que se ejecute el método *bindService()* en la *Activity*. La **¡Error! No se ncuentra el origen de la referencia.** muestra el código implementado.

```
public class LocalBinder extends Binder {
    ConnectService getService() {
        return ConnectService.this;
    }
}

private final IBinder mBinder = new LocalBinder();

@Override
public IBinder onBind(Intent intent) {
    return mBinder;
};
```

Figura 41. Código para vincular con la Activity

Respecto a la conexión con el dispositivo, se ha implementado dos métodos distintos (Figura 42), uno para establecer la conexión, que se ha llamado *connect()* y otro para deshacerla, denominado *disconnect()*. El funcionamiento de cada método es el siguiente:

- ***connect()*:** en primer lugar recuperará y almacenará en un objeto *BluetoothDevice* el dispositivo al que deseamos conectarnos, para ello empleará el método *getRemoteDevice()*, al que pasaremos la dirección del

dispositivo. Una vez realizado, se establecerá la conexión utilizando las funciones *connectGatt()*, propio de la clase *BluetoothGatt* y *connect()*, propio de la clase *BluetoothDevice*, que además comprobarán que la conexión se ha llevado a cabo correctamente.

- ***disconnect()*:** únicamente llamará al método propio de la clase *BluetoothDevice disconnect()* que cortará la conexión.

```
public boolean connect(final String address) {
    Log.i(TAG, "Start");
    if (BLeAdapter == null || address == null) {
        Log.w(TAG,
                "BluetoothAdapter not initialized or unspecified address.");
        return false;
    }

    BLeDevice = BLeAdapter.getRemoteDevice(address);
    BLeGatt = BLeDevice.connectGatt(this, false, BLeGattCallback);
    if (BLeGatt.connect())
        Log.i(TAG, "Connected");
    else
        Log.i(TAG, "Not Connected");
    DeviceAddress = address;
    ConnectionState = STATE_CONNECTING;
    return true;

}

public void disconnect(){
    BLeGatt.disconnect();
}
```

Figura 42. Código para establecer y cortar la conexión con el dispositivo

Por último se ha implementado un método que se encargará de emitir los anuncios que se recibirán en la *Activity* a la que está vinculado el *Service*, esta función se ha denominado *broadcastUpdate()*. Para ello hemos creado un objeto de la clase *BluetoothGattCallback* que contiene diversos métodos que serán llamados según los cambios que se produzcan en la conexión:

***onConnectionStateChange()* (Figura 43):** este método será llamado cuando se produzca un cambio en el estado de la conexión con el dispositivo, es decir, conectado o desconectado. Se encargará de emitir el anuncio correspondiente a cada estado.

```
@Override
public void onConnectionStateChange(BluetoothGatt gatt, int status,
        int newState) {
    if (newState == BluetoothProfile.STATE_CONNECTED) {
        intentAction = ACTION_GATT_CONNECTED;
        ConnectionState = STATE_CONNECTED;
        broadcastUpdate(intentAction);
        Log.i(TAG, "Connected to GATT server.");
        Log.i(TAG,
                "Attempting to start service discovery:"
                        + BLeGatt.discoverServices());
    }
    if (newState == BluetoothProfile.STATE_DISCONNECTED) {
        intentAction = ACTION_GATT_DISCONNECTED;
        ConnectionState = STATE_DISCONNECTED;
        broadcastUpdate(intentAction);
        Log.i(TAG, "Disconnected from GATT server.");
    }
}
```

Figura 43. Código onConnectionStateChange

- **onServicesDiscovered() (Figura 44):** se llamará a este método en el momento que se descubran los servicios del dispositivo remoto. Si se descubren con éxito se emitirá el anuncio correspondiente.

```
@Override
public void onServicesDiscovered(BluetoothGatt gatt, int status) {
    if (status == BluetoothGatt.GATT_SUCCESS) {
        Log.i(TAG, "onServicesDiscovered GattSuccess");
        intentAction = ACTION_GATT_SERVICES_DISCOVERED;
        broadcastUpdate(intentAction);
    } else {
        Log.i(TAG, "onServicesDiscovered received: " + status);
    }
}
```

Figura 44. Código onServicesDiscovered

- ***onCharacteristicRead()* (Figura 45):** cuando se lea correctamente el valor de una característica este método será llamado, se encargará de enviar el anuncio que corresponda.

```
@Override
public void onCharacteristicRead(BluetoothGatt gatt,
        BluetoothGattCharacteristic characteristic, int status) {
    if (status == BluetoothGatt.GATT_SUCCESS) {
        intentAction = ACTION_DATA_AVAILABLE;
        final Sensor<?> sensor = SensorTag.getSensor(characteristic
                .getService().getUuid().toString());
        Log.i(TAG, "onCharacteristicRead: " + sensor.getName());
        if (sensor != null)
            if (sensor.onCharacteristicRead(characteristic))
                return;
        broadcastUpdate(intentAction, characteristic);
    }
}
```

Figura 45. Código onCharacteristicRead

- ***onCharacteristicChanged()*** **(Figura 46):** este método se llamará cuando se produzca algún cambio en el valor de alguna de las características y emitirá el anuncio correspondiente.

```
@Override
public void onCharacteristicChanged(BluetoothGatt gatt,
        BluetoothGattCharacteristic characteristic) {
    intentAction = ACTION_DATA_AVAILABLE;
    broadcastUpdate(intentAction, characteristic);
}
```

Figura 46. Código onCharacteristicChanged

- ***onCharacteristicWrite()*** **(Figura 47):** este último método será llamado cuando escribamos en el valor de alguna característica. Emitirá el anuncio que le corresponda.

```
@Override
public void onCharacteristicWrite(BluetoothGatt gatt,
        BluetoothGattCharacteristic characteristic, int status) {
    if(characteristic.getUuid().equals(PressureSensor.UUID_CALIBRATION))
        return;
    Sensor<?> sensor = SensorTag.getSensor(characteristic.getService()
            .getUuid().toString());
    Log.i(TAG, "onCharacteristicWrite: " + sensor.getName());
    sensor.notify(BLeGatt, true);
    readCharacteristic(characteristic);
}
```

Figura 47. Código onCharacteristicWrite

A continuación, en los apartados que siguen, se desarrollará la explicación sobre la sección de la aplicación que se encargará de interactuar con el controlador del sistema.

12.4.2.6 Device Selection Activity

A partir de este apartado comenzaremos con la explicación de la sección de la aplicación que se encargará de conectar con el sistema de control del aire acondicionado y el humidificador.

En este primer apartado explicaremos la implementación llevada a cabo para poder seleccionar un dispositivo de control y conectar con él empleando el *Service Bluetooth Service.* En la Figura 48 se muestra el diseño de la actividad.

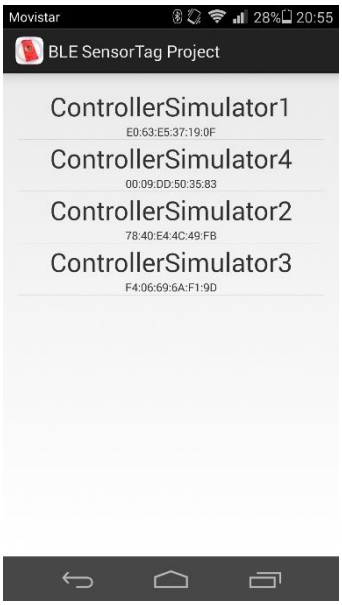

Figura 48. Captura de pantalla de Device Selection Activity

Esta *Activity*, al igual que sucedía con la actividad para seleccionar una *SensorTag* y la *Main Activity*, se diseñado sobre una *ListActivity* en la que se mostrarán los dispositivos disponibles para establecer conexión. Para que un dispositivo pueda aparecer en esta lista previamente tendrá que haber sido emparejado con el dispositivo móvil del usuario, puesto que únicamente se mostrarán en la lista dichos dispositivos.

En primer lugar, al iniciar la actividad, se iniciará también el *Bluetooth Service* empleando para ello el método *startService()*, una vez en iniciado lo vincularemos a la actividad por medio del método *bindService()*. El código se muestra en la siguiente figura.

```
setListAdapter(devicesList);
btAdapter.cancelDiscovery();
btAdapter.startDiscovery();
findDevices();
final Intent ServiceIntent = new Intent(this, BluetoothService.class);
startService(ServiceIntent);
bindService(ServiceIntent, ServiceConnection, BIND_AUTO_CREATE);
```

Figura 49. Código para iniciar y vincular el Service *e iniciar el descubrimiento de dispositivos*

Para almacenar los dispositivos disponibles crearemos, tal y como hicimos previamente, un objeto de la clase *Device List Adapter*, al que denominaremos *devicesList*. Como se muestra en la Figura 49, primero deberemos comenzar el descubrimiento de dispositivos utilizando el método *startDiscovery()*. Una vez iniciado, llamaremos al método *findDevices()* (Figura 50), el cual se encargará de buscar todos los dispositivos emparejados y añadirlos a *devicesList*. Esta lista se mostrará en la pantalla empleando el método, ya mencionado anteriormente, *setListAdapter()*.

```
private void findDevices() {
    devicesList.clear();
    pairedDevices = btAdapter.getBondedDevices();
    // If there are paired devices
    if (pairedDevices.size() > 0) {
        // Loop through paired devices
        for (BluetoothDevice device : pairedDevices) {
            // Add the name and address to an array adapter to show in a
            // ListView
            devicesList.addDevice(device);
        }
    }
}
```

Figura 50. Código para buscar y añadir los dispositivos a la lista

Una vez se muestran todos los dispositivos disponibles para establecer la conexión, el usuario seleccionará el dispositivo con el que quiere conectar. Cuando pulsa sobre el dispositivo escogido se llama al método del *Bluetooth Service connect()*, que se encargará de establecer la conexión el mismo.

Además, deberemos crear también un *Broadcast Receiver* (Figura 51) para gestionar los anuncios emitidos por el *Service*. A diferencia del *Connect Service* el *Bluetooth*

Service únicamente emitirá un anuncio para indicar que la conexión se ha llevado a cabo correctamente o que no se ha podido establecer conexión:

- Para el caso de que la conexión se haya realizado correctamente, se iniciará una nueva actividad, llamada *Change Values Activity*, para ello volveremos a emplear un *Intent*, en el que, al igual que en *SensorTag Selection Activity*, introduciremos el nombre y la dirección del dispositivo con el que hemos conectado y posteriormente llamaremos al método *startActivity()* e iniciaremos la nueva actividad.

- En el otro caso, cuando no ha sido posible conectar con el dispositivo, se lo indicaremos al usuario empleando un *Toast*, o mensaje emergente, en el que indicaremos que la conexión no ha podido ser establecida y permaneceremos en esta actividad.

```
final BroadcastReceiver receiver = new BroadcastReceiver() {

    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        Log.i(TAG, action);
        if (action.equals("UNABLE_TO_CONNECT")) {
            Toast.makeText(getBaseContext(),
                    "Unnable to Connect to: " + btDevice.getName(),
                    Toast.LENGTH_LONG).show();
        }
        if(action.equals("CONNECTED")){
            Toast.makeText(getBaseContext(),
                    "Connected to: " + btDevice.getName(),
                    Toast.LENGTH_LONG).show();
            Intent myIntent = null;
            myIntent = new Intent(getBaseContext(), ChangeValuesActivity.class);
            myIntent.putExtra(ChangeValuesActivity.EXTRAS_DEVICE_NAME,
                    btDevice.getName());
            myIntent.putExtra(ChangeValuesActivity.EXTRAS_DEVICE_ADDRESS,
                    btDevice.getAddress());
            startActivity(myIntent);
        }

    }
};
```

Figura 51. Código del Broadcast Receiver

Por último, cuando la actividad es destruida se realiza la desvinculación del *Service.* Además, en esta *Activity*, también lo detendremos empleando el método *stopService()*. El código empleado se muestra en la Figura 52.

```
@Override
protected void onDestroy() {
    super.onDestroy();
    Log.i(TAG, "Destroy");
    unregisterReceiver(receiver);
    unbindService(ServiceConnection);
    final Intent ServiceIntent = new Intent(this, BluetoothService.class);
    stopService(ServiceIntent);
}
```

Figura 52. Código para desvincular y detener el Service

A continuación procederemos con la explicación de la siguiente actividad que utilizaremos para configurar los sistemas de climatización. También será la última de la aplicación.

12.4.2.7 Change Values Activity

En último lugar tendremos la actividad *Change Values Activity*, en la cual introduciremos la configuración del sistema que queremos y que, a través del *Bluetooth Service*, pasaremos al dispositivo de control que configurarán los distintos elementos que forman el sistema, esto es, el aire acondicionado y el humidificador. En la siguiente figura se muestra el diseño escogido.

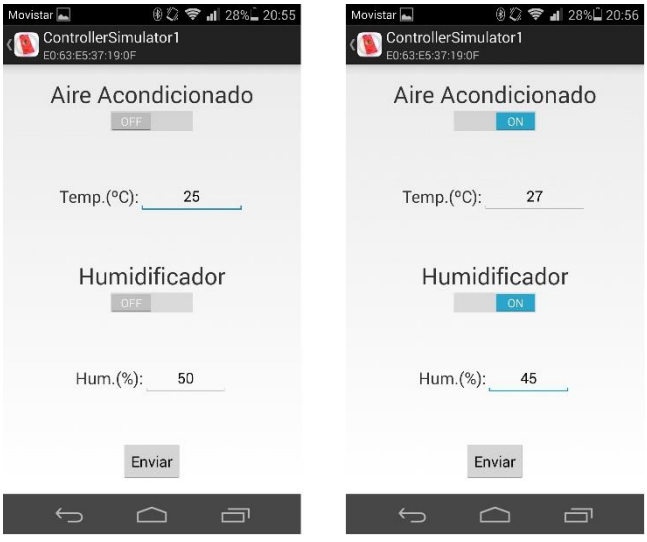

Figura 53. Captura de pantalla de Change Values Activity

Al iniciarse la actividad, del mismo modo que en actividades anteriores, se ha realizado la vinculación con el *Bluetooth Service*.

El diseño de la interfaz de esta *Activity* se ha realizado en tres secciones distintas, dos de ellas son simétricas puesto que son donde introduciremos los datos de los sistemas de climatización y la última de ellas consiste en el botón que emplearemos para enviar los datos al controlador de los sistemas. En la Figura 54 se muestra la declaración de los distintos elementos que forman la interfaz.

La estructura de las dos primeras secciones es la siguiente:

- En primer lugar tendremos un cuadro de texto (*TextView*) que mostrará cuál de los dos sistemas está configurando el usuario, es decir, el aire acondicionado o el humidificador. Este cuadro de texto es un elemento fijo y el usuario no podrá ni modificarlo ni interactuar con él.

- En segundo lugar tendremos un interruptor (*Switch*) que se utilizará para que el usuario indique si desea activar o desactivar el sistema. Predeterminadamente se encontrará desactivado y deberá ser el usuario quien lo active.

- Por último tendremos otro cuadro de introducción de texto (*EditText*) en el que el usuario introducirá el valor de temperatura o humedad, según corresponda, al que queremos configurar el sistema. Este cuadro inicialmente tendrá un valor intermedio, 25ºC para el aire acondicionado y 45% para el humidificador. El valor introducido no será enviado al controlador si el interruptor no se encuentra activado.

```
EditText Temperature;
EditText Humidity;
Switch HVAC;
Switch Humid;
Button btnSend;
```

Figura 54. Declaración de los elementos de la interfaz gráfica

La última sección, como ya se ha indicado anteriormente, está formado únicamente por un botón. Cuando se pulse el botón los datos introducidos en los distintos campos de la interfaz serán enviados utilizando el método del *Bluetooth Service write()*, que se llama desde un hilo distinto del principal ya que este método bloque el hilo en el cual se implementa. Sin embargo, antes de enviarlos será necesario juntarlos todos en una sola variable de texto (*String*), que es el parámetro recibido por el método *write()* ya

mencionado. En la Figura 55 y la Figura 56 se muestra el código implementado para el envío de los datos.

```
btnSend.setOnClickListener(new View.OnClickListener() {

    @Override
    public void onClick(View v) {
        SendThread1.send(getString());
    }
});
```

Figura 55. Código para envío al pulsar Enviar

```
private class SendThread extends Thread {

    public SendThread() {

    }

    public void run() {

    }

    public void send(String msg) {
        if (mBound) {
            btService.mConnectedThread.write(msg);
        }
    }

}
```

Figura 56. Código para evitar el bloqueo del hilo principal de la aplicación al enviar los datos

En primer lugar se llamará a los métodos *getTemperature()* y *getHumidity()* que juntará los datos introducidos en un *String*, indicando el estado de activación del interruptor y en el caso de que esté activado añadirá también el valor de configuración introducido por el usuario. Seguidamente, se usará el método *sendData()* que combinará los *String* de ambos sistemas, aire acondicionado y humedad, en uno solo para poder enviarlo. En la Figura 57 se muestra el código que transformará los datos introducidos por el usuario para poder enviarlos.

```
protected String getTemperature() {
    String HVACinfo = "";
    if (HVAC.isChecked()) {
        HVACinfo = "AC: ON \t" + "T: " + Temperature.getText().toString();
    } else {
        HVACinfo = "AC: OFF";
    }
    return HVACinfo;
}

protected String getHumidity() {
    String HumidInfo = "";
    if (Humid.isChecked())
        HumidInfo = "HUM: ON \t" + "H: " + Humidity.getText().toString();
    else
        HumidInfo = "HUM: OFF \t";
    return HumidInfo;
}

protected String sendData() {
    String sOut = "";
    sOut = getTemperature() + "\n" + getHumidity();
    return sOut;
}
```

Figura 57. Código para obtener la información introducida

Por último, al igual que el resto de actividades vinculadas a un *Service*, al destruir la *Activity* está se desvinculará del *Service.* Además, se desconectará del controlador empleando el método del *Bluetooth Service disconnect()* y se guardará la configuración introducida por el usuario empleando un objeto de la clase *SharedPreferences*.

Seguidamente, en el siguiente apartado, continuaremos con la implementación desarrollada para el *Bluetooth Service.*

12.4.2.8 Bluetooth Service

Finalmente, procederemos con la explicación de la última parte del bloque principal de la aplicación, el *Bluetooth Service*, que será el encargado de establecer y gestionar la conexión con el dispositivo controlador del sistema que emplee el usuario. Al igual que en el caso de anterior *Service*, el *Connect Service*, carece de interfaz gráfica con la que el usuario pueda interactuar puesto que se ejecutará en segundo plano.

Como ya hemos explicado en el anterior *Service*, en primer lugar se han implementado los métodos necesarios para para poder vincularlo con la *Activity*, mostrados en la Figura 58.

```
public class LocalBinder extends Binder {
    BluetoothService getService() {
        return BluetoothService.this;
    }
}

private final IBinder mBinder = new LocalBinder();

@Override
public IBinder onBind(Intent intent) {
    return mBinder;
};
```

Figura 58. Código para vincular con la Activity

Para la gestión de los distintos estados de la conexión, esto es, cuando se está realizando la conexión y cuando los dispositivos ya está conectados, se han implementado dos *Threads*, o hilos separados del principal, para evitar el bloqueo de este puesto que los métodos empleados para gestionar la conexión bloquean el hilo en el que son llamados hasta que han finalizado su trabajo. Los *Threads* creados son los siguientes:

- ***ConnectThread*:** este *Thread* es el encargado de iniciar la conexión con el dispositivo. En él se implementará un método denominado *run()* (Figura 60), que será el que será llamado cuando se inicie el hilo, dentro se llamará al método propio de la clase *BluetoothSocket connect()*, que establecerá la conexión con el dispositivo remoto. Además, también se implementará el método *cancel()* (Figura 59), que llamaremos cuando queramos detener el *Thread*, y que cortará la conexión utilizando otro método de la clase *BluetoothSocket close()*. Una vez la conexión se haya establecido se llamará a la función *connected()*, que explicaremos más adelante.

```
/** Will cancel an in-progress connection, and close the socket */
public void cancel() {
    try {
        mmSocket.close();
    } catch (IOException e) {
    }
}
```

Figura 59. Código para cerrar el hilo y la conexión

```
public void run() {
    // Cancel discovery because it will slow down the connection
    Log.i(TAG, "mmSocket.connect(1)");
    btAdapter.cancelDiscovery();

    Log.i(TAG, "mmSocket.connect(2)");

    try {
        // Connect the device through the socket. This will block
        // until it succeeds or throws an exception
        Log.i(TAG, "mmSocket.connect(3)");
        mmSocket.connect();
    } catch (IOException connectException) {
        // Unable to connect; close the socket and get out
        Log.w(TAG, "Unable to Connect");
        intentAction = "UNABLE_TO_CONNECT";
        broadcastUpdate(intentAction);
        try {
            mmSocket.close();
        } catch (IOException closeException) {
        }
        return;
    }
```

Figura 60. Código para iniciar el hilo y la conexión

- ***ConnectedThread*:** este hilo se ha creado para manejar la conexión una vez ha sido establecida. En este caso no se implementará nada dentro del método *run()*, sino que se implementará una nueva función, denominada *write()* que se encargará de realizar el envío de los datos a través de la conexión, para ello, dentro de este método se llamará a la función propia del clase *OutputStream write()* que será la que envíe los datos. En este caso, al igual que en el anterior también se ha implementado el método *cancel()* para cortar la conexión y cerrar el hilo. En la Figura 61 se muestra el código para el envío de datos.

```
/* Call this from the main activity to send data to the remote device */
public void write(String msg) {
    if (mmSocket.isConnected()) {
        Log.i(TAG, "Socket Connected");
        try {
            msg += "\n";
            Log.i(TAG, msg + msg.getBytes().length);
            mmOutStream.write(msg.getBytes());
        } catch (IOException e) {
            Log.i(TAG, "WRITE ERROR");
        }
    } else
        Log.w(TAG, "Socket Not Connected");
}
```

Figura 61. Código para envío de datos a través de la conexión

Además de estos dos hilos, se han implementado otros tres métodos que serán los encargados de iniciar y cerrar los hilos ya explicados, que serán los encargados de

establecer y cortar la conexión y enviar los datos al dispositivo remoto. Estos métodos son los siguientes:

- ***connect()* (Figura 62):** en esta función iniciaremos el *ConnectThread*. Para ello previamente deberemos haber comprobado que no ha sido iniciado previamente. En la figura **x** se muestra el código implementado.

```
public boolean connect(BluetoothDevice device) {
    Log.i(TAG, "Initialazing connection...");
    if (mConnectThread != null) {
        mConnectThread.cancel();
        mConnectThread = null;
    }
    btDevice = device;
    mConnectThread = new ConnectThread(device);
    mConnectThread.start();
    Log.i(TAG, "Initialazing finished...");
    return true;
}
```

Figura 62. Código para iniciar ConnectThread

- ***connected()* (Figura 63):** con este método iniciaremos el *ConnectedThread*. En este caso también comprobaremos si ha sido iniciado previamente. En la siguiente figura se muestra el código.

```
public void connected(BluetoothSocket socket) {
    if (mConnectedThread != null) {
        mConnectedThread.cancel();
        mConnectedThread = null;
    }

    mConnectedThread = new ConnectedThread(socket);
    mConnectedThread.start();
}
```

Figura 63. Código para iniciar ConnectedThread

- **disconnect() (Figura 64):** este método detendrá todos los Threads que hayan sido abiertos previamente. En esta figura se muestra el código del método.

```
public void disconnect() {
    if (mConnectThread != null)
        mConnectThread.cancel();
    if (mConnectedThread != null)
        mConnectedThread.cancel();
}
```

Figura 64. Código para cerrar los Threads *iniciados*

Con esto finaliza la explicación del bloque principal de la aplicación. A lo largo de los siguientes apartados se explicarán el resto de clases necesaria para el correcto funcionamiento de la aplicación.

12.4.2.9 Paquete Adapters

A continuación procederemos con la explicación de las clases creadas para poder realizar la lista en las actividades *SensorTag Search Activity, Device Selection Activity* y *Sensor Read Activity*.

12.4.2.9.1 Clase *Device List Adapter*

En este apartado explicaremos la implementación llevada a cabo para el *Adapter* que emplearemos para almacenar los dispositivos encontrados por Bluetooth.

Para poder almacenar y posteriormente mostrar los dispositivos es necesario asignarle un *Layout* (Figura 65) con la estructura en la que se mostrarán los datos. Esta estructura estará formada por dos partes distintas, aunque ambas serán cuadros de texto:

- El primero de los cuadros mostrará el nombre del dispositivo almacenado.
- En el segundo de ellos se mostrará la dirección MAC del dispositivo.

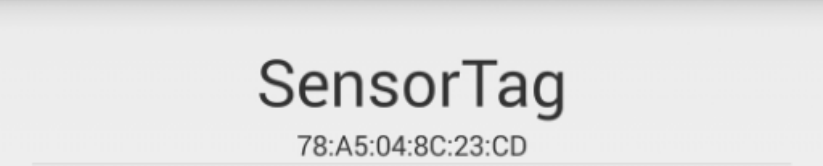

Figura 65. Captura de pantalla del Layout *de los dispositivos* SensorTag

Además del *Layout*, también hemos implementado diferentes métodos que serán necesarios para poder interactuar con la información almacenada en el *Adapter*. Dichos métodos son los siguientes:

- ***DeviceListAdapter()*:** este método será el constructor que emplearemos para crear el *Adapter* en la *Activity*.
- ***addDevices()*:** este método nos permitirá añadir nuevos dispositivos a la lista.
- ***clear()*:** con este método borraremos toda la información que hayamos almacenado.
- ***getDevice()*:** utilizado para obtener alguno de los dispositivos almacenados. Será principalmente utilizado cuando el usuario haya seleccionado un dispositivo en la *Activity*.

En el siguiente apartado desarrollaremos el otro *Adapter* implementado.

12.4.2.9.2 Clase *Sensors List Adapter*

A lo largo de este apartado se describirá como se ha implementado el *Adapter* para almacenar los sensores que están disponibles para mostrar datos, en nuestro caso filtraremos para mostrar únicamente los sensores de humedad y temperatura.

En este caso, al igual que el otro *Adapter*, también deberemos asignar un *Layout* (Figura 66) para mostrar los datos almacenados. El diseño será el mismo que en caso anterior, esto es, con dos cuadros de texto, el primero mostrará el nombre del sensor almacenado, sin embargo, el segundo no mostrará la dirección sino que mostrará un número de identificación denominado UUID.

Figura 66. Captura de pantalla del Layout *de los sensores*

Los métodos implementados para este *Adapter*, son similares a los del caso anterior. Son los siguientes:

- ***SensorsListAdapter()*:** este método será el constructor, en el que además se filtrarán los sensores para que únicamente se añadan el de humedad y temperatura.

- ***addServices()*:** con este método podremos añadir nuevos sensores a la lista después de haber utilizado el constructor.
- ***getService()*:** método utilizado para obtener uno de los sensores almacenados en el *Adapter*.
- ***clear()*:** nos permitirá borrar todo el contenido almacenado.

En el siguiente apartado y los posteriores se desarrollarán las clases necesarias para gestionar los sensores.

12.4.2.10 Paquete SensorTag

A lo largo de los siguientes apartados explicaremos la implementación llevada a cabo en las clases necesarias para la gestión de los sensores que emplearemos con esta aplicación.

12.4.2.10.1 Clase *SensorTag*

Esta primera clase la emplearemos para almacenar en un *HashMap* los datos de los sensores que vamos a utilizar en la aplicación. La diferencia con la clase *Sensors List Adapter* es que en esta clase los sensores los añade el programador de la aplicación, mientras que en el *Adapter* se añaden en tiempo de ejecución cuando son descubiertos el móvil.

Se ha implementado además un método para recuperar los datos del sensor, lo hemos denominado *getSensor()*, y devuelve un objeto de la clase *Sensor*, la cual explicaremos a continuación. La Figura 67 muestra el código implementado.

```
public class SensorTag {
    private static HashMap<String, Sensor<?>> SENSORS = new HashMap<String, Sensor<?>>();

    static {
        final TemperatureSensor temperatureSensor = new TemperatureSensor();
        final HumiditySensor humiditySensor = new HumiditySensor();

        SENSORS.put(temperatureSensor.getServiceUUID(), temperatureSensor);
        SENSORS.put(humiditySensor.getServiceUUID(), humiditySensor);
    }

    public static Sensor<?> getSensor(String uuid) {
        return SENSORS.get(uuid);
    }

}
```

Figura 67. Código clase SensorTag

Seguidamente explicaremos la clase donde incluiremos los métodos necesarios para gestionar los sensores.

12.4.2.10.2 Clase *Sensor*

En esta sección desarrollaremos la clase en la cual hemos incluido todos los métodos necesarios para poder manejar los sensores remotamente y obtener los datos de temperatura y humedad. Sin embargo, la clase ha sido implementada como abstracta, por lo tanto, no se podrán crear objetos de la misma y muchos de los métodos únicamente serán declarados y la implementación de los mismos se realizara en las clases particulares de cada sensor, puesto que es distinta para cada uno de los sensores.

En primer lugar se realizará una descripción de los métodos declarados en esta clase, pero que serán implementados en otras:

- ***getServiceUUID()*:** con este método recuperaremos el UUID de servicio (sensor).
- ***getDataUUID()*:** usado para recuperar los el UUID de la característica en la que se almacena el valor del dato que lee el sensor, esto es, la humedad o la temperatura.
- ***getConfigUUID()*:** método empleado para obtener el UUID de la característica donde se incluye el valor de la configuración del sensor, que en nuestro caso será si está activo o no.
- ***getStringData()*:** este método permitirá introducir los datos recibidos del sensor y almacenados en su característica dentro de un *String* que será el que mostremos al usuario.
- ***getName()*:** con este método recuperaremos el nombre del sensor.
- ***parse()*:** emplearemos este método para descodificar el dato recibido del sensor de forma que obtengamos un valor que pueda ser entendido por el usuario de la aplicación.

Además de todos estos métodos abstractos, también se han implementado diversos métodos que son comunes para todos los sensores utilizados. Estos métodos son los siguientes:

- ***enableSensor()*:** método usado para activar el sensor puesto que estos se encuentran predeterminadamente desactivados.
- ***read()*:** con este método leeremos el valor almacenado en la característica que hayamos seleccionado.
- ***write()*:** este método nos permitirá escribir un nuevo valor dentro de la característica. Lo usaremos para escribir el valor para activar el sensor en su respectiva característica.
- ***notify()*:** usaremos este método para establecer las notificaciones cuando el valor de una característica concreta varíe. Nos servirá para actualizar la pantalla con el nuevo valor cada vez que este cambie.
- ***shortSignedAtOffset()* y *shortUnsignedAtOffset()* (Figura 68):** estos dos métodos serán los utilizados para transformar los datos, que se reciben en formato de bytes, en variables decimales que podrán ser transformados por el método *parse()* para finalmente mostrárselos al usuario. Este código lo proporciona el proveedor.
- ***getCharacteristic()*:** con este método obtendremos una de las características del sensor, para ello deberemos pasarle el UUID de la característica que queramos.

A continuación se explicarán las clases particulares de cada sensor, comenzando por el de humedad.

```
public static Integer shortUnsignedAtOffset(BluetoothGattCharacteristic c,
        int offset) {
    Integer lowerByte = c.getIntValue(FORMAT_UINT8, offset);
    if (lowerByte == null)
        return 0;
    Integer upperByte = c.getIntValue(FORMAT_UINT8, offset + 1); // Note:
                                                                 // interpret
                                                                 // MSB
                                                                 // as
                                                                 // unsigned.
    if (upperByte == null)
        return 0;

    return (upperByte << 8) + lowerByte;
}

public static Integer shortSignedAtOffset(BluetoothGattCharacteristic c,
        int offset) {
    Integer lowerByte = c.getIntValue(FORMAT_UINT8, offset);
    if (lowerByte == null)
        return 0;
    Integer upperByte = c.getIntValue(FORMAT_SINT8, offset + 1); // Note:
                                                                 // interpret
                                                                 // MSB
                                                                 // as
                                                                 // signed.
    if (upperByte == null)
        return 0;

    return (upperByte << 8) + lowerByte;
}
```

Figura 68. Código para obtener decimales a partir de los bytes recibidos

12.4.2.10.3 Clase *Humidity Sensor*

En este apartado se explicará la implementación llevada a cabo para el sensor de humedad. Se han implementado todos los métodos que se habían declarado en la clase madre, es decir, la clase *Sensor*, con los datos del sensor de humedad. En la Figura 69 se muestra el código necesario para decodificar los datos recibidos del sensor, el cual es proporcionado por el proveedor en la guía de usuario.

```
@Override
protected Float parse(BluetoothGattCharacteristic c) {
    int a = shortUnsignedAtOffset(c, 2);
    a = a - (a % 4);
    Float b = (-6f) + 125f * (a / 65535f);
    if (b < 60 && b > 30) {
        Hum.setTextColor(Color.parseColor("#00FF00"));
        Hum.setBackgroundColor(Color.parseColor("#006600"));
    } else {
        Hum.setTextColor(Color.parseColor("#FF9900"));
        Hum.setBackgroundColor(Color.parseColor("#FF0000"));
    }
    return b;
}
```

Figura 69. Código para decodificar la humedad

12.4.2.10.4 Clase *Temperature Sensor*

Por último realizaremos la explicación de la última clase incluida en la aplicación. En ella se han implementado, al igual que en la clase *Humidity Sensor*, los métodos declarados en la clase madre con los datos del sensor de temperatura. Así mismo, ha sido necesario la implementación de dos nuevos métodos para decodificar los datos, puesto que el sensor de temperatura proporciona dos valores. El primero de ellos es la temperatura del ambiente, que se visualiza como "Ambiente", y el segundo de ellos la del sensor de infrarrojos, que depende de la temperatura ambiente y que se mostrará como "Temperatura IR". Los métodos implementados son los siguientes:

- ***extractAmbientTemperature()* (Figura 70):** este método decodificará el dato de la temperatura de boquilla enviado por el sensor.

```
private static double extractAmbientTemperature(
        BluetoothGattCharacteristic c) {
    int offset = 2;
    return shortUnsignedAtOffset(c, offset) / 128.0;
}
```

Figura 70. Código para decodificar temperatura ambiente

- **extractTargetTemperature() (Figura 71):** con este método decodificaremos el dato de la temperatura obtenida por el infrarrojo.

```
private static double extractTargetTemperature(
        BluetoothGattCharacteristic c, double ambient) {
    Integer twoByteValue = shortSignedAtOffset(c, 0);

    double Vobj2 = twoByteValue.doubleValue();
    Vobj2 *= 0.00000015625;

    double Tdie = ambient + 273.15;

    double S0 = 5.593E-14; // Calibration factor
    double a1 = 1.75E-3;
    double a2 = -1.678E-5;
    double b0 = -2.94E-5;
    double b1 = -5.7E-7;
    double b2 = 4.63E-9;
    double c2 = 13.4;
    double Tref = 298.15;
    double S = S0 * (1 + a1 * (Tdie - Tref) + a2 * pow((Tdie - Tref), 2));
    double Vos = b0 + b1 * (Tdie - Tref) + b2 * pow((Tdie - Tref), 2);
    double fObj = (Vobj2 - Vos) + c2 * pow((Vobj2 - Vos), 2);
    double tObj = pow(pow(Tdie, 4) + (fObj / S), .25);

    return tObj - 273.15;
}
```

Figura 71. Código para decodificar temperatura del infrarrojo

Los datos de estas funciones se utilizarán en la función *parse()*, mostrada en la Figura 72 y se juntarán en un vector decimal.

```
@Override
public float[] parse(BluetoothGattCharacteristic c) {
    double ambient = extractAmbientTemperature(c);
    double target = extractTargetTemperature(c, ambient);
    if (ambient > 30) {
        Temp.setTextColor(Color.parseColor("#FF9900"));
        Temp.setBackgroundColor(Color.parseColor("#FF0000"));
    }else if(ambient<20){
        Temp.setTextColor(Color.parseColor("#00FFFF"));
        Temp.setBackgroundColor(Color.parseColor("#0000FF"));
    }else{
        Temp.setTextColor(Color.parseColor("#00FF00"));
        Temp.setBackgroundColor(Color.parseColor("#006600"));
    }

    return new float[] { (float) ambient, (float) target };
}
```

Figura 72. Código para juntar ambas temperaturas

Con esto finaliza la explicación de la implementación llevada a cabo para la interfaz gráfica de usuario, es decir, la aplicación Android.

13 PRUEBAS

A continuación se procederá con la explicación de las pruebas realizadas para comprobar el correcto funcionamiento y el cumplimiento de los objetivos puestos al proyecto. Parte de las pruebas realizadas a la aplicación se han realizado de forma paralela a la implementación puesto que como parte del desarrollo es necesario probar las modificaciones que se van realizando al código, dichas pruebas serán las de funcionamiento y comunicación. Así mismo, una vez finalizado el desarrollo y habiéndose comprobado el correcto funcionamiento de la aplicación completa, se ha procedido con la realización de las pruebas de cobertura y precisión del sistema.

13.1 Pruebas de funcionamiento

Las pruebas de funcionamiento de la aplicación se han realizado un vez completado el diseño visual de la misma y de forma paralela a la implementación del código. Para ello se ha empleado el Smartphone personal del desarrollador, un Huawei Ascend P7, en el que se ha ido instalando al aplicación cada vez que se realizaban modificaciones sobre la misma y comprobando el correcto funcionamiento. Una vez solucionados los errores encontrados, se ha repetido el proceso de prueba y corrección de errores hasta haber finalizado completamente el desarrollo de la interfaz.

Tras haber completado la implementación de la interfaz se ha procedido con su instalación en diversos dispositivos móviles distintos al empleado para el desarrollo y se ha comprobado su correcto funcionamiento y que las proporciones visuales de la interfaz se mantenían. Estas pruebas se han realizado en los dispositivos de la Tabla 3.

DISPOSITIVO	HUAWEI ASCEND P7	SAMSUNG GALAXY CORE PRIME	SONY XPERIA E1
PANTALLA (PULG.)	5	4,5	4
RESOLUCIÓN (PIXELES)	1080 x 1920	480 x 800	480 x 800
BLUETOOTH 4.0	SI	SI	SI

Tabla 3. Móviles Android empleados en las pruebas

Como se puede observar se trata de móviles con tamaños de pantalla y resoluciones diferentes, por lo tanto serán ideales para comprobar que la aplicación se adapta correctamente a distintas pantallas y resoluciones. De la Figura 73 a la Figura 75 se muestran las capturas de pantalla de la actividad *Change Values Activity*, que debido a que es la que más elementos visuales contiene y permitirá una mejor comprobación.

Figura 73. Huawei Ascend P7

Figura 74. Samsung Galaxy Core Prime

Figura 75. Sony Xperia E1

A continuación se procederá con la explicación de las pruebas de comunicación entre los distintos dispositivos realizadas.

13.2 Pruebas de comunicación

En cuanto a las pruebas de comunicación, al igual que las de funcionamiento, se han realizado simultáneamente al desarrollo de la aplicación para poder resolver los errores que pudiesen surgir al realizarlas. Sin embargo, las pruebas de comunicación se han dividido en dos partes, una primera durante el desarrollo de la *Activity Sensor Read* y la segunda durante la implementación de la actividad *Device Selection Activity*. En la primera de ellas comprobaremos que la conexión con el dispositivo *SensorTag* se realiza correctamente y que una vez seleccionado un sensor se reciben los datos, tal y como se muestra en las figuras **X y X**.

En el caso de la segunda prueba, también se comprobará la correcta conexión con el controlador y que el envío de los datos se realiza correctamente. Sin embargo, debido a que no disponemos de dicho controlador, hemos empleado otros dispositivos como simuladores, dos portátiles, Lenovo G50-80 y Asus S300C, con los programas *Hyperterminal* y *TeraTerm* instalados, que mostrarán los datos recibidos del móvil, y dos móviles, Samsung Galaxy Core Prime y Sony Xperia E1, para los que se ha implementado una aplicación ex profeso que se encargará de recibir y mostrar los

datos. De la Figura 77 a la Figura 80 se pueden ver los resultados de las pruebas de comunicación realizadas, con diferentes combinaciones de configuración.

Seguidamente, en el siguiente apartado, procederemos con las pruebas de cobertura del sistema.

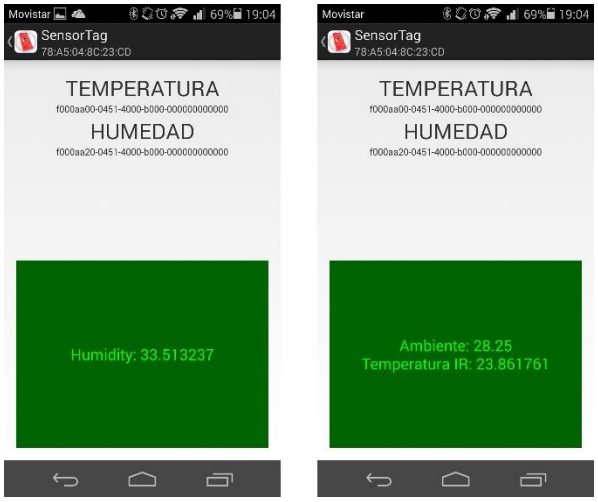

Figura 76. Pruebas de comunicación con el SensorTag

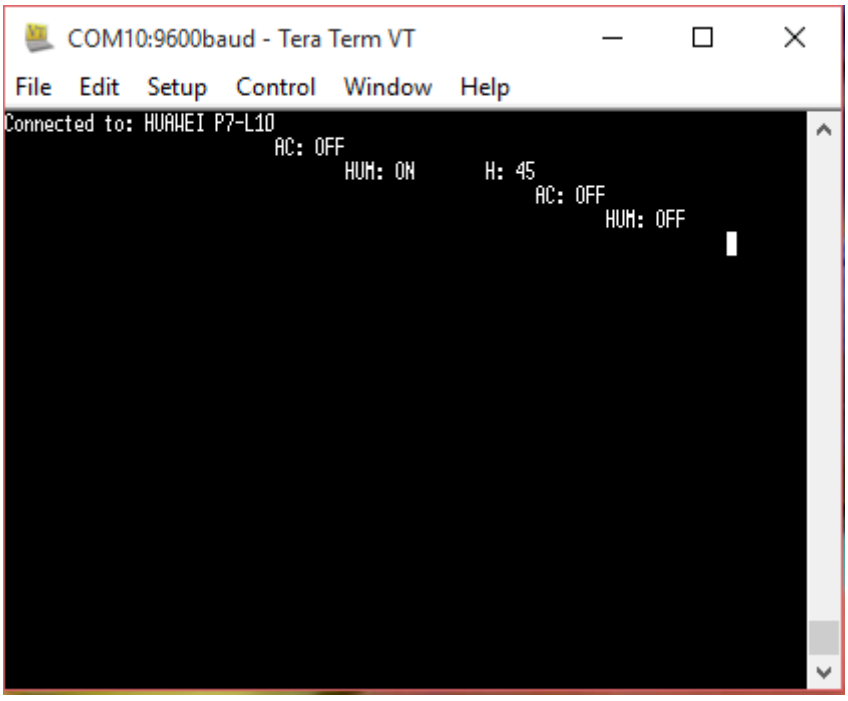

Figura 77. Prueba comunicación Lenovo + TeraTerm

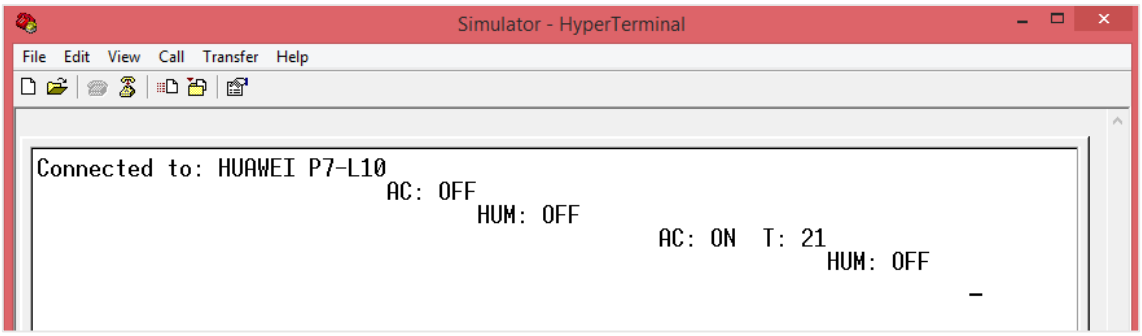

Figura 78. Prueba comunicación ASUS + Hyperterminal

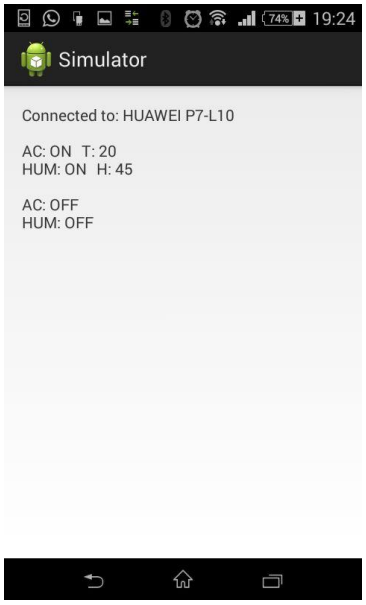

Figura 79. Prueba Sony Xperia E1

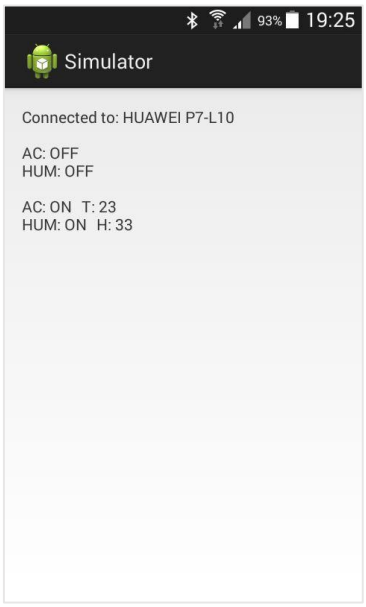

Figura 80. Prueba Samsung Galaxy Core Prime

13.3 Pruebas de cobertura

Tras haber comprobado el correcto funcionamiento de la aplicación y su correcta comunicación con los distintos dispositivos empleados, se ha procedido a realizar una prueba de la cobertura. Para ello se ha empleado la vivienda del desarrollador, puesto que al tratarse de una aplicación para usos domóticos, está enfocada a su uso en estructuras domésticas con numerosos obstáculos que podrían reducir la cobertura.

Para la realización de la prueba, se ha situado en un extremo de la vivienda un *SensorTag* y un dispositivo que actúe como el simulador del controlador, tal y como se ha explicado en el apartado anterior. Seguidamente el desarrollador comienza a establecer conexión, cada vez a una distancia superior y con mayor número de obstáculos intermedios, comprobando en cada momento que la conexión se realiza exitosamente y que la transmisión de datos es correcta.

Finalmente, se ha comprobado la comprobado que la conexión se establece correctamente en toda la vivienda, siendo mejor la cobertura de la tecnología Low Energy.

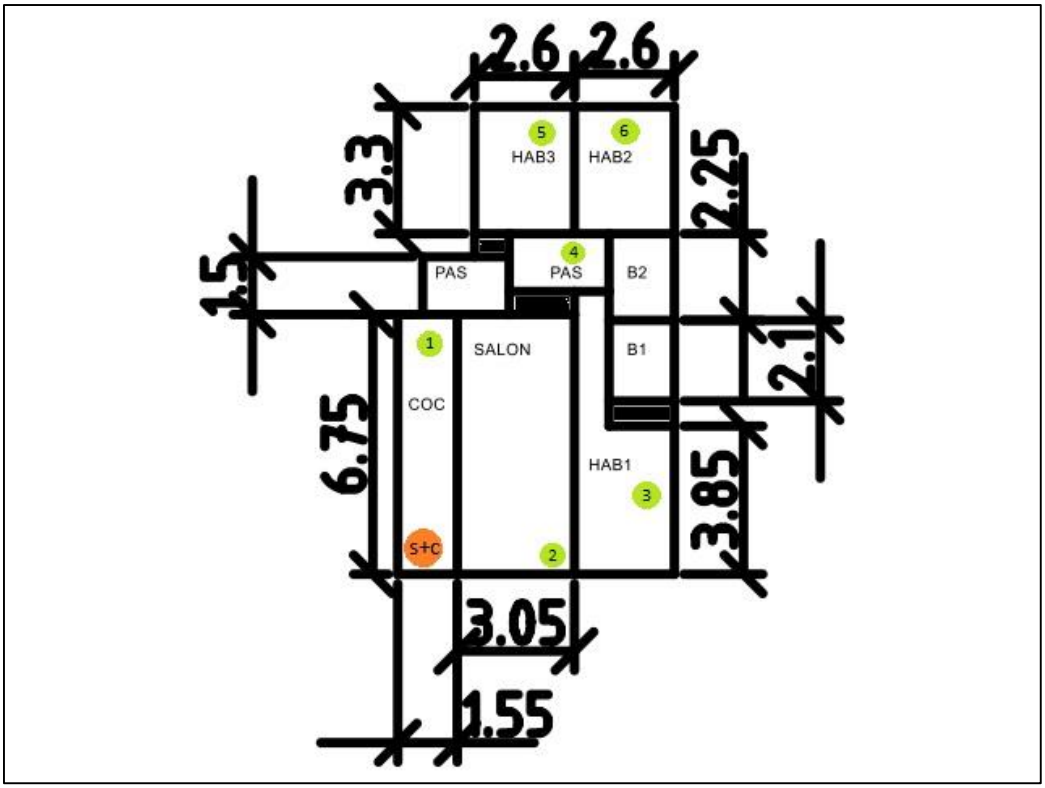

Figura 81. Plano de pruebas de cobertura

Como se puede observar en la Figura 81, se ha situado el *SensorTag* y el simulador en el punto marcado de rojo y se ha comprobado la comunicación en cada uno de los 6 puntos marcados de verde, fallando únicamente la comunicación en el punto 6, con lo que la cobertura es adecuada para una vivienda de tamaño medio.

13.4 Pruebas de precisión

Para realizar la prueba de fiabilidad de los sensores se ha decidido comparar los datos proporcionados por estos, con los proporcionados con una estación meteorológica en el mismo punto. Para determinar que los datos recibidos de los sensores son correctos, se han establecido los siguientes criterios según el sensor que se esté probando:

Temperatura: Desviación máxima entre el dato de la estación y el sensor de temperatura de ±3ºC. Obtenido de la suma de los márgenes de error de los dos dispositivos, ±2ºC de la estación y ±1ºC del sensor para la temperatura ambiente.

Humedad: Desviación máxima entre el dato de la estación y el sensor de humedad de un 6%. Obtenido de la suma de los márgenes de error de ambos dispositivos, que son de un 4% para la estación y de un 2% para el sensor.

A continuación en la Figura 82 y la Figura 83 se muestran los resultados de las pruebas de precisión realizadas.

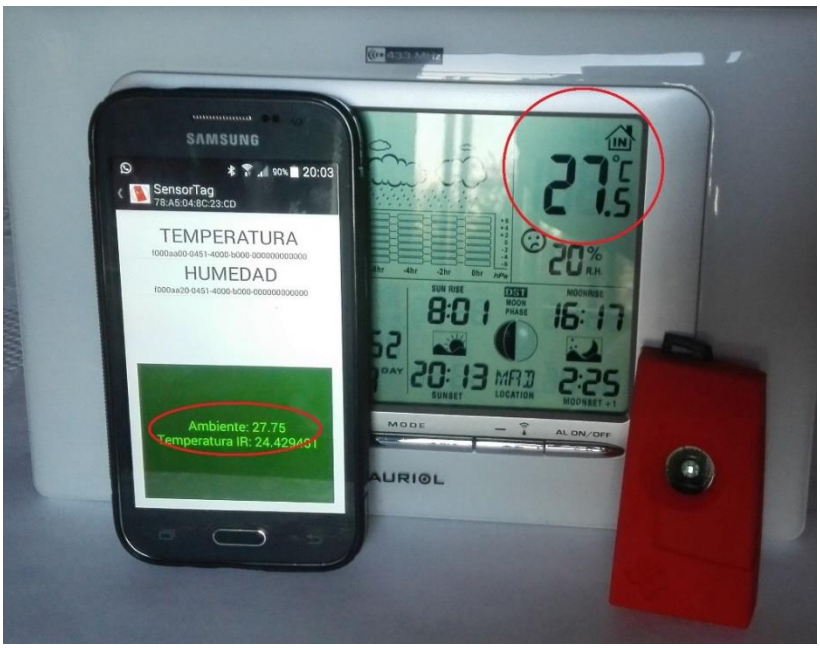

Figura 82. Prueba de precisión del sensor de temperatura

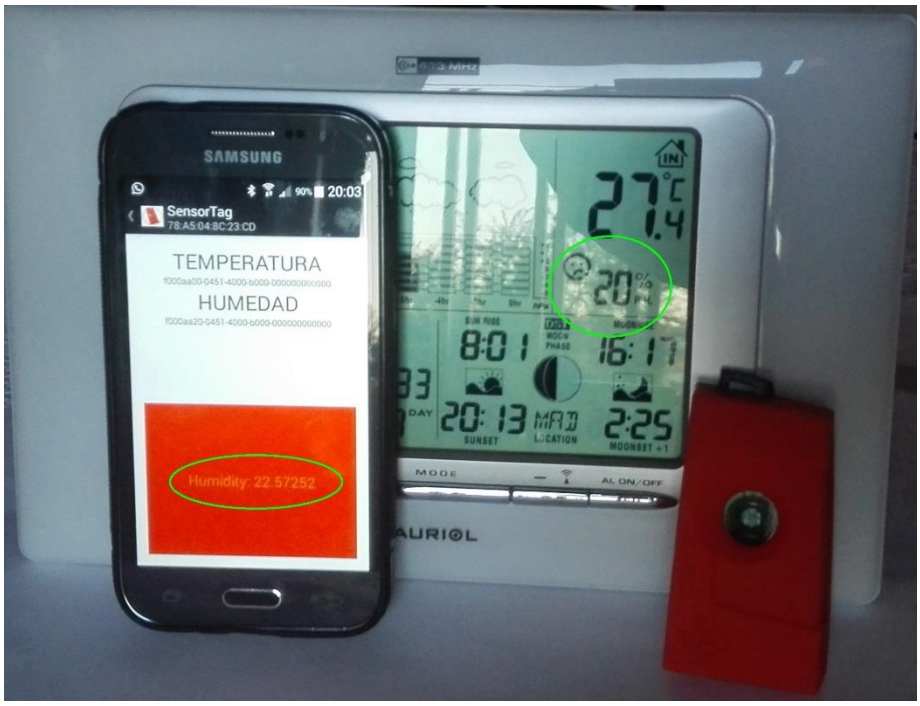

Figura 83. Prueba de precisión de la humedad

14 CONCLUSIONES

El objetivo planteado en el momento de la elección de este proyecto fue la creación de una aplicación que permitiese a su usuario la conexión con un sistema domótico, en este caso de climatización, y a partir de los datos proporcionados por los sensores, leídos por la aplicación, configurar dicho sistema con los parámetros que el usuario desee. Finalmente, se han podido llevar a cabo satisfactoriamente todos los objetivos de menor escala en los que se ha dividido el principal, mejorando las expectativas esperadas en el momento del planteamiento del proyecto.

En el caso de la parte de la aplicación encargada de la recepción de los datos de los sensores, se han llevado a cabo con éxito tanto la conexión con el dispositivo *SensorTag*, como la lectura y actualización en tiempo real de los datos transmitidos por el mismo. Además, como añadido se le ha implementado un sistema que es capaz de indicar al usuario si los valores transmitidos por el sensor son adecuados para una vivienda.

Por otro lado, para la parte que se encargará de la comunicación con el controlador del sistema con el que se emplee la aplicación desarrollada, también se han cumplido ambos objetivos planteados ya que tanto la conexión como el envío de los datos se produce correctamente. Así mismo, se ha resuelto el problema de la falta de disponibilidad de controlador con el que conectarse, puesto que se trata únicamente del desarrollo de la aplicación de gestión, empleando como simulador un PC u otro dispositivo Android con una aplicación diseñada ex profeso.

Por último, la realización de este proyecto también ha cumplido con otro de los objetivos, y motivo por el cual se ha decido realizarlo, ya que ha sido necesario aprender un nuevo lenguaje de programación, que además está en alza en la actualidad y que proporcionará un valor añadido al desarrollador en una futura búsqueda de empleo.

15 TRABAJOS FUTUROS

El proyecto realizado deja la puerta abierta a la continuación de su desarrollo. A continuación se desarrollan diversas posibilidades, que podrían realizarse para mejorar las características del proyecto:

- **Desarrollo de aplicación para dispositivos iOS:** pese a que, como se ha explicado anteriormente en este documento, se ha escogido usar Android como sistema operativo de desarrollo, es innegable el gran número de dispositivos que emplean este sistema. Por tanto, el desarrollo de una aplicación para sistema iOS, ampliaría en gran medida la cantidad de dispositivos en los que la aplicación podría ser utilizada.

- **Comunicación con varios dispositivos simultáneamente:** otra posible mejora sobre el proyecto desarrollado, sería modificar la aplicación de manera que nos permitiese la conexión simultánea con varios dispositivos, ya fuesen *SensorTag* o controladores de sistema, que posibilitaría la visualización de la temperatura y humedad de varias habitaciones al mismo tiempo y, también, ajustar los valores de configuración de varios sistemas sin tener que conectarse y modificarlos uno por uno.

- **Implementación del controlador del sistema:** otra de las mejoras aplicables sobre este proyecto, consistiría en la programación de un microprocesador o microcontrolador que actuaría como dispositivo controlador de los sistemas de climatización que programaría dichos sistemas con los datos introducidos en la aplicación desarrollada.

- **Ampliar la cobertura del sistema:** finalmente, empleando la tecnología de redes de datos o la tecnología WI-FI para realizar la conexión de los distintos elementos del sistema a gran distancia permitiría la monitorización y ajuste desde puntos alejados de la vivienda donde se encuentre instalado el sistema gestionado por la aplicación.

16 GESTIÓN DEL PROYECTO

A lo largo de esta sección se desarrollará la gestión del proyecto llevada a cabo, se indicará la planificación de las distintas fases del proyecto y los costes en los que se ha incurrido para la realización de este proyecto.

16.1 Planificación

En este apartado se mostrarán las distintas actividades y tareas en las que se ha dividido la realización del proyecto para alcanzar todos los objetivos planteados. En la Tabla 4 se representan las actividades generales del proyecto.

ACTIVIDAD	FECHA INICIO	FECHA FIN
1. Análisis del proyecto	**26/01/2015**	**22/03/2015**
2. Formación	**09/02/2015**	**12/04/2015**
3. Diseño de la interfaz	**16/03/2015**	**12/04/2015**
4. Implementación	**13/04/2015**	**30/08/2015**
5. Pruebas	**02/06/2015**	**13/09/2015**
6. Redacción de la memoria	**15/07/2015**	**24/09/2015**

Tabla 4. Resumen de actividades del proyecto

A continuación (Tabla 5) se mostrará la planificación detallada y un cronograma (Figura 84) de las actividades con las distintas tareas que se realizarán dentro de las mismas.

ACTIVIDAD/TAREA	FECHA DE INICIO	FECHA DE FIN
1. Análisis del proyecto	**26/01/2015**	**22/03/2015**
1.1. Estudio de posibilidades del proyecto	26/01/2015	01/02/2015
1.2. Decisiones de diseño	04/02/2015	22/02/2015
1.3. Búsqueda de información	04/02/2015	22/03/2015
2. Formación	**09/02/2015**	**12/04/2015**
2.1. Java	09/02/2015	22/02/2015
2.2. Android	23/02/2015	12/04/2015
3. Diseño de la interfaz	**16/03/2015**	**12/04/2015**
4. Implementación	**13/04/2015**	**30/08/2015**
4.1. Implementación de la interfaz	13/04/2015	30/08/2015
4.2. Corrección de errores de la interfaz	02/06/2015	30/08/2015
4.3. Modificaciones sobre el diseño original	03/08/2015	30/08/2915
5. Pruebas	**03/08/2015**	**13/09/2015**
5.1. Pruebas de funcionamiento	02/06/2015	30/08/2015
5.2. Pruebas de comunicación	06/07/2015	30/08/2015
5.3. Pruebas de cobertura	31/08/2015	06/09/2015
5.4. Pruebas de precisión	07/09/2015	13/09/2015
6. Redacción de la memoria	**15/07/2015**	**24/09/2015**

Tabla 5. Planificación detallada del proyecto

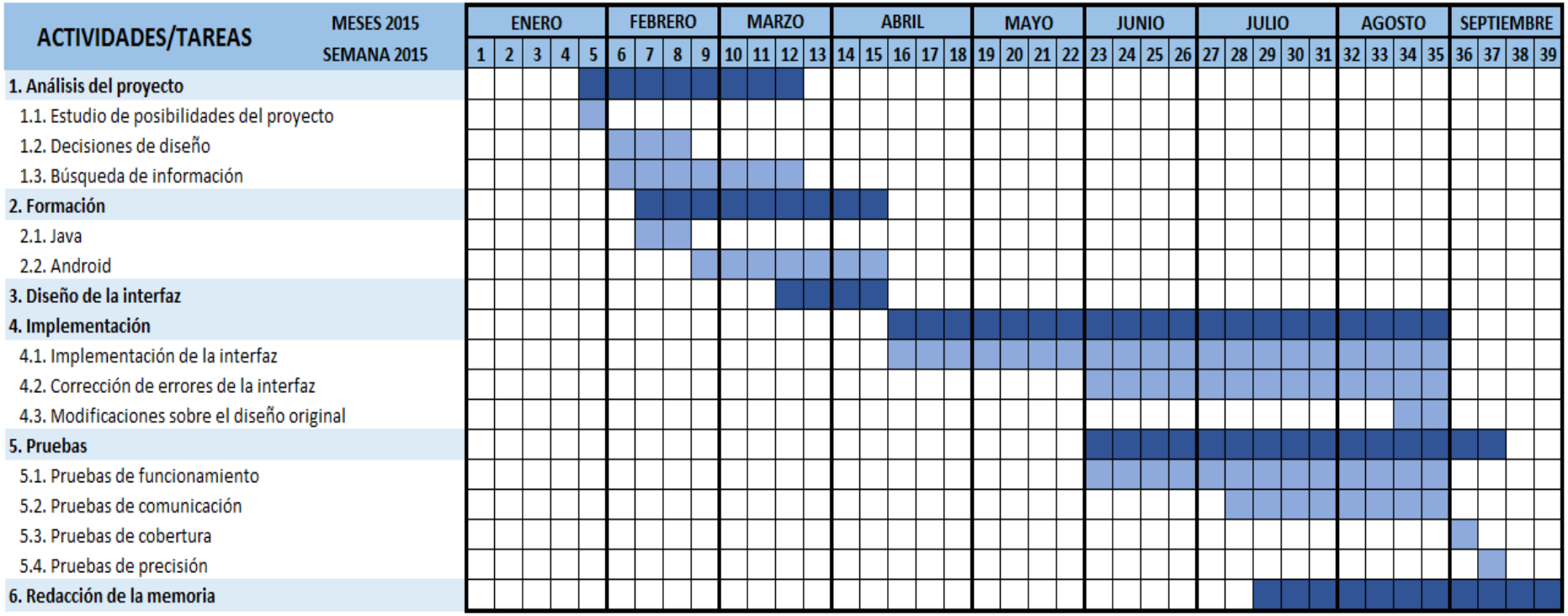

Figura 84. Cronograma detallado de la planificación del proyecto

16.2 Presupuesto

Para realizar el cálculo del presupuesto se ha dividido en tres partes:

- **Coste de personal:** incluye todo el coste de las horas empleadas por las personas encargadas de la realización del proyecto.
- **Costes de material de laboratorio:** incluye todos los costes del material empleado para el desarrollo y prueba del proyecto.
- **Coste del hardware:** costes del hardware utilizado para el desarrollo del proyecto.

16.2.1 Coste de personal

Los costes de personal del proyecto se incluyen en la Tabla 6**.** Para su cálculo se ha tenido en cuenta el tiempo empleado para el desarrollo del proyecto y el salario medio de un ingeniero junior con 1 año de experiencia en España **[23]**.

NOMBRE	PERFIL	DURACIÓN (MESES)	SALARIO (€/MES)	TOTAL (€)
Ruiz Izquierdo, Raúl	Ingeniero Junior	7	1600	11200
			TOTAL	**11200,00**

Tabla 6. Costes de personal

16.2.2 Costes del material de laboratorio

Para los costes de laboratorio se ha empleado el coste de amortización de cada uno de los equipos empleados para el desarrollo. Dicho coste se ha calculado empleando la siguiente fórmula:

$$C.A = \frac{M}{T.A} \times C \times U \qquad (7)$$

Donde:

- C.A: Coste de amortización.
- M: Tiempo de utilización del equipo durante el proyecto.

- T.A: Tiempo de amortización del equipo.
- C: Coste del equipo
- U: % de uso del equipo dado al proyecto (habitualmente 100%).

Los costes del material de laboratorio se detallan en la Tabla 7.

EQUIPO	UTILIZACIÓN (MESES)	UTILIZACIÓN (%)	COSTE (€)	TIEMPO AMORTIZACIÓN (MESES)	COSTE AMORTIZACIÓN (€)
Portátil ASUS S300C	7	100	599	60	69,88
Portátil Lenovo G50-80	2	100	399	60	13,30
Huawei Ascend P7	5	100	260	60	21,67
Samsung Galaxy Core Prime	2	100	129	60	4,30
Sony Xperia E1	2	100	89	60	2,97
				TOTAL	**112,12**

Tabla 7. Coste del material de laboratorio

16.2.3 Coste de hardware

En la Tabla 8 se muestra los costes del hardware empleado en el proyecto.

COMPONENTE	CANTIDAD	COSTE UNITARIO (€)	COSTE (€)
CC2541 SensorTag Development Kit	2	24,99	49,98
		TOTAL	**49,98**

Tabla 8. Costes de hardware

16.2.4 Resumen de costes

Finalmente, en la Tabla 9 se muestra el resumen de todos los costes incurridos durante la realización del proyecto, además de añadir el IVA, que será de un 21%, sobre el coste total incurrido.

DESCRIPCIÓN	COSTE (€)
Costes de personal	11200,00
Costes del material de laboratorio	112,12
Costes de hardware	49,98
SUBTOTAL	11362,10
IVA (21%)	2386,04
TOTAL	**13748,14**

Tabla 9. Resumen de costes del proyecto

17 REFERENCIAS

[1] Qué es Domótica. [Online] http://www.cedom.es/sobre-domotica/que-es-domotica [Julio 2015]

[2] Historia de la Domótica. [Online] http://www.arkiplus.com/historia-de-la-domotica [Julio 2015]

[3] Sistema de Comunicación Domótico X-10 [Online] http://www.monografias.com/trabajos99/sistema-domotico-x-10/sistema-domotico-x-10.shtml [Julio 2015]

[4] MOYA SÁNCHEZ, Alejandro, 2012. *Control domótico de las plazas de un aparcamiento*. Manuel Antolín Arias, dir. Proyecto Fin de Carrera. Universidad Carlos III de Madrid, Departamento de Ingeniería Eléctrica, Leganés. [Online] http://hdl.handle.net/10016/16828 [Julio 2015]

[5] VEGA, Ricardo. Introducción a KNX y topología. En: *ricveal.com* [Online] http://www.ricveal.com/knx-intro/ [Julio 2015]

[6] VEGA, Ricardo. Direccionamiento y Acceso al medio KNX. En: *ricveal.com* [Online] http://www. ricveal.com/direccionamiento-acceso-knx/ [Julio 2015]

[7] VEGA, Ricardo. Telegramas en KNX. En: *ricveal.com* [Online] http://www. ricveal.com/telegramas-knx/ [Julio 2015]

[8] VEGA, Ricardo. Ventajas e inconvenientes de EIB KONNEX. En: *ricveal.com* [Online] http://www. ricveal.com/ventajas-inconvenientes-eib-konnex/ [Julio 2015]

[9] Bluetooth. [Online] http://es.ccm.net/contents/70-bluetooth [Agosto 2015]

[10] HUNN, Nick, 08/07/2010. Bluetooth. HUNN, Nick. *Essential of Short-Range Wireless.* Cambridge: Cambridge University Press, 34. 978-0-521-76069-0. [Online] http://proquest.safaribooksonline.com.strauss.uc3m.es:8080/9780511771354 [Agosto 2015]

[11] Como funciona Bluetooth. [Online] http://es.ccm.net/contents/69-como-funciona-bluetooth [Agosto 2015]

[12] SensorTag User Guide. [Online] http://processors.wiki.ti.com/index.php/SensorTag_User_Guide [Septiembre 2015]

[13] TMP006 Contactless IR Temperature Sensor, Datasheet, Texas Instruments. [Online] http://www.ti.com/lit/ds/symlink/tmp006.pdf [Septiembre 2015]

[14] TMP006 Contactless IR Temperature Sensor, User Guide, Texas Instruments. [Online] http://www.ti.com/lit/ug/sbou107/sbou107.pdf [Septiembre 2015]

[15] SHT21 Humidity Sensor, Datasheet, Sensirion. [Online] http://www.sensirion.com/fileadmin/user_upload/customers/sensirion/Dokumente/Humidity/Sensirion_Humidity_SHT21_Datasheet_V4.pdf [Septiembre 2015]

[16] SHT2x - Digital Humidity & Temperature Sensor (RH/T), Sensirion. [Online] http://www.sensirion.com/en/products/humidity-temperature/humidity-temperature-sensor-sht2x/ [Septiembre 2015]

[17] Android Developer. [Online] http://developer.android.com/develop/index.html [Septiembre 2015]

[18] International Data Corporation, IDC. [Online] http://www.idc.com/prodserv/smartphoneos-market-share.jsp [Septiembre 2015]

[19] Android Activity. [Online] http://developer.android.com/training/basics/activitylifecycle/index.html [Septiembre 2015]

[20] Android Service. [Online] http://developer.android.com/guide/components/services.html [Septiembre 2015]

[21] Android Bluetooth. [Online] http://developer.android.com/guide/topics/connectivity/bluetooth.html [Septiembre 2015]

[22] Android Bluetooth Low Energy. [Online] http://developer.android.com/guide/topics/connectivity/bluetooth-le.html [Septiembre 2015]

[23] Referencia sueldo ingeniero. [Online]

http://www.pagepersonnel.es/content/salary-research.htm [Septiembre 2015]

18 ANEXOS

18.1 Diagramas de flujo de la aplicación

De la Figura 85 a la **¡Error! No se encuentra el origen de la referencia.** se muestran los diagramas de flujo de las distintas actividades implementadas en la aplicación, además del ciclo de vida de la aplicación completa

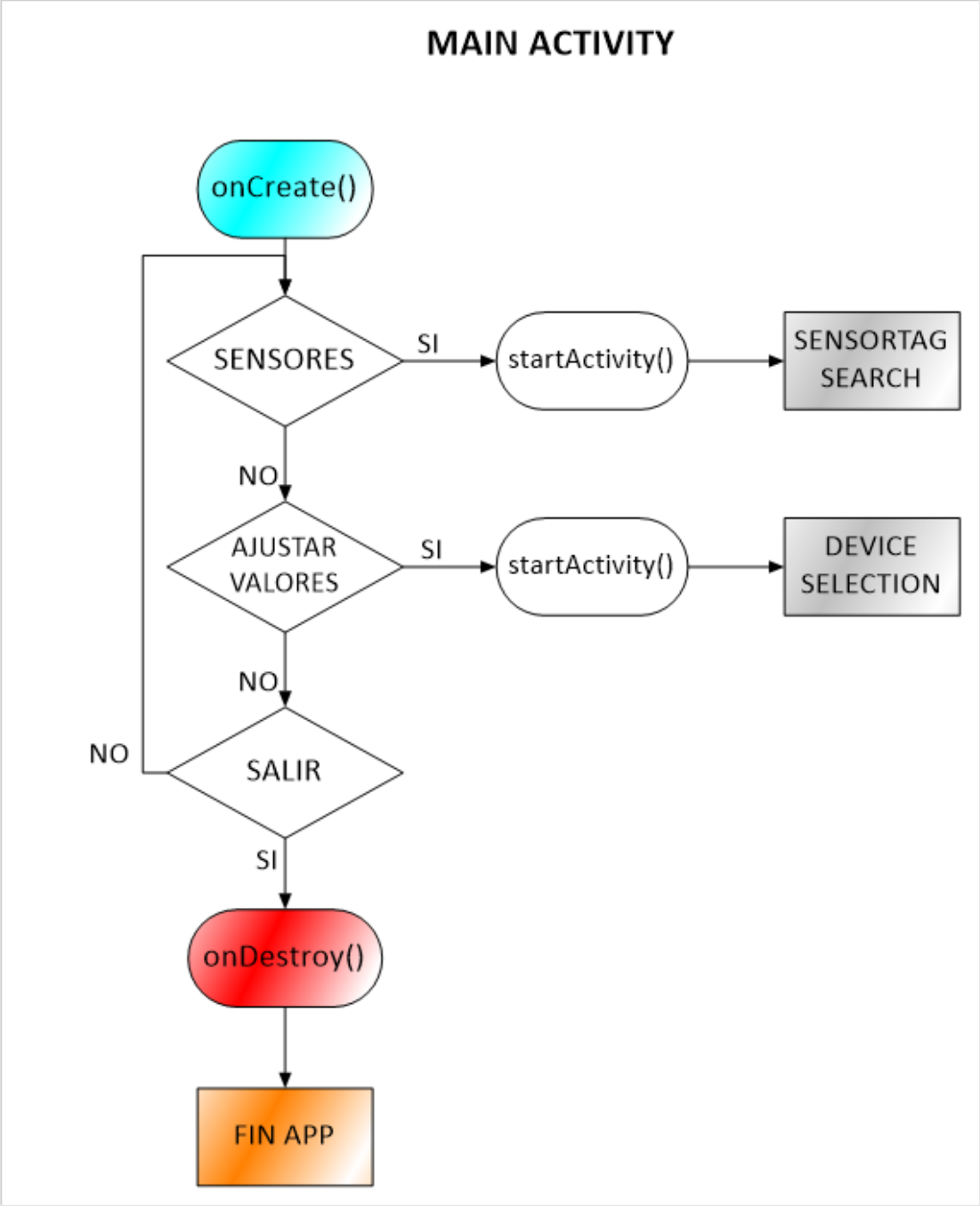

Figura 85. Diagrama de flujo de Main Activity

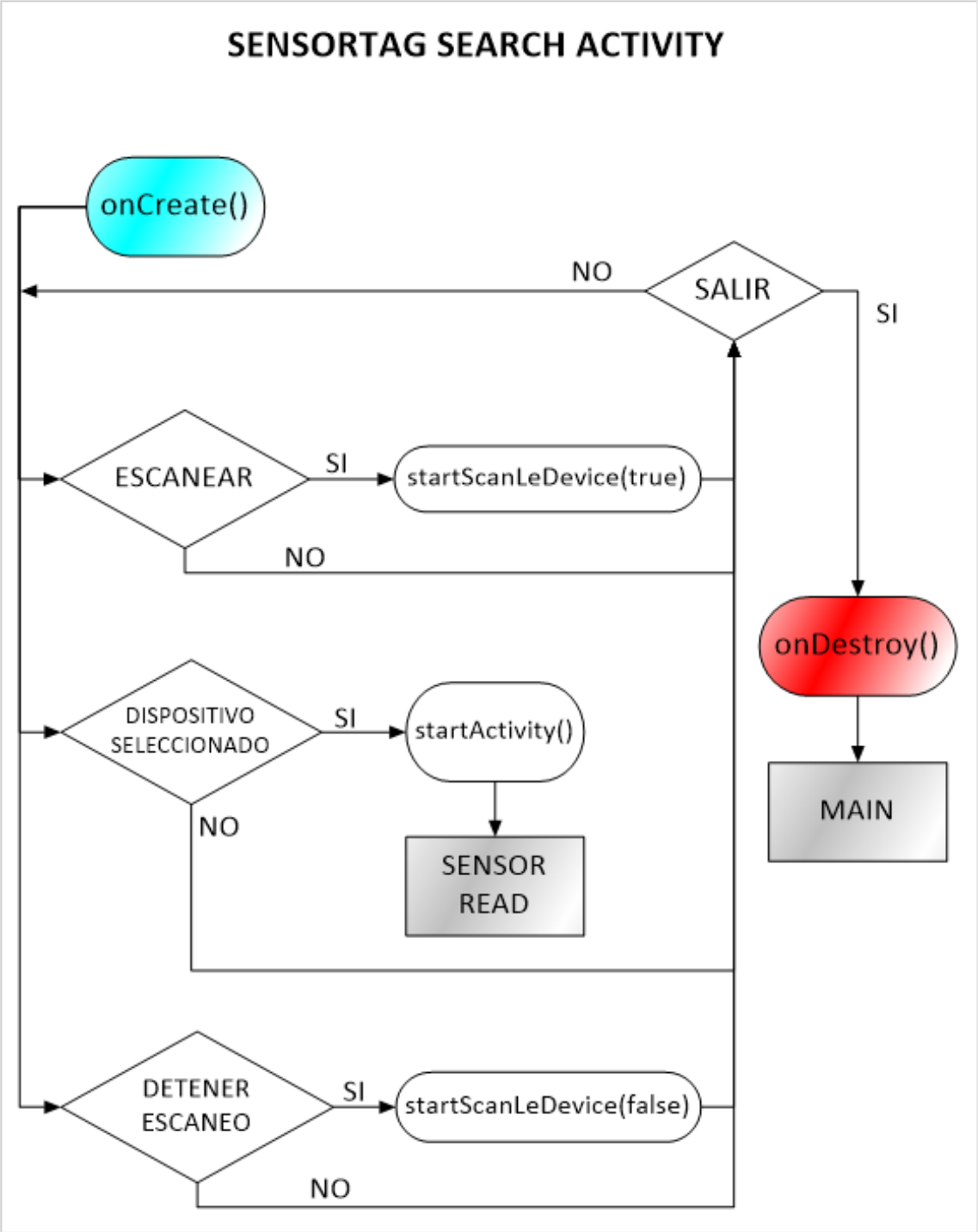

Figura 86. Diagrama de flujo de SensorTag Search Activity

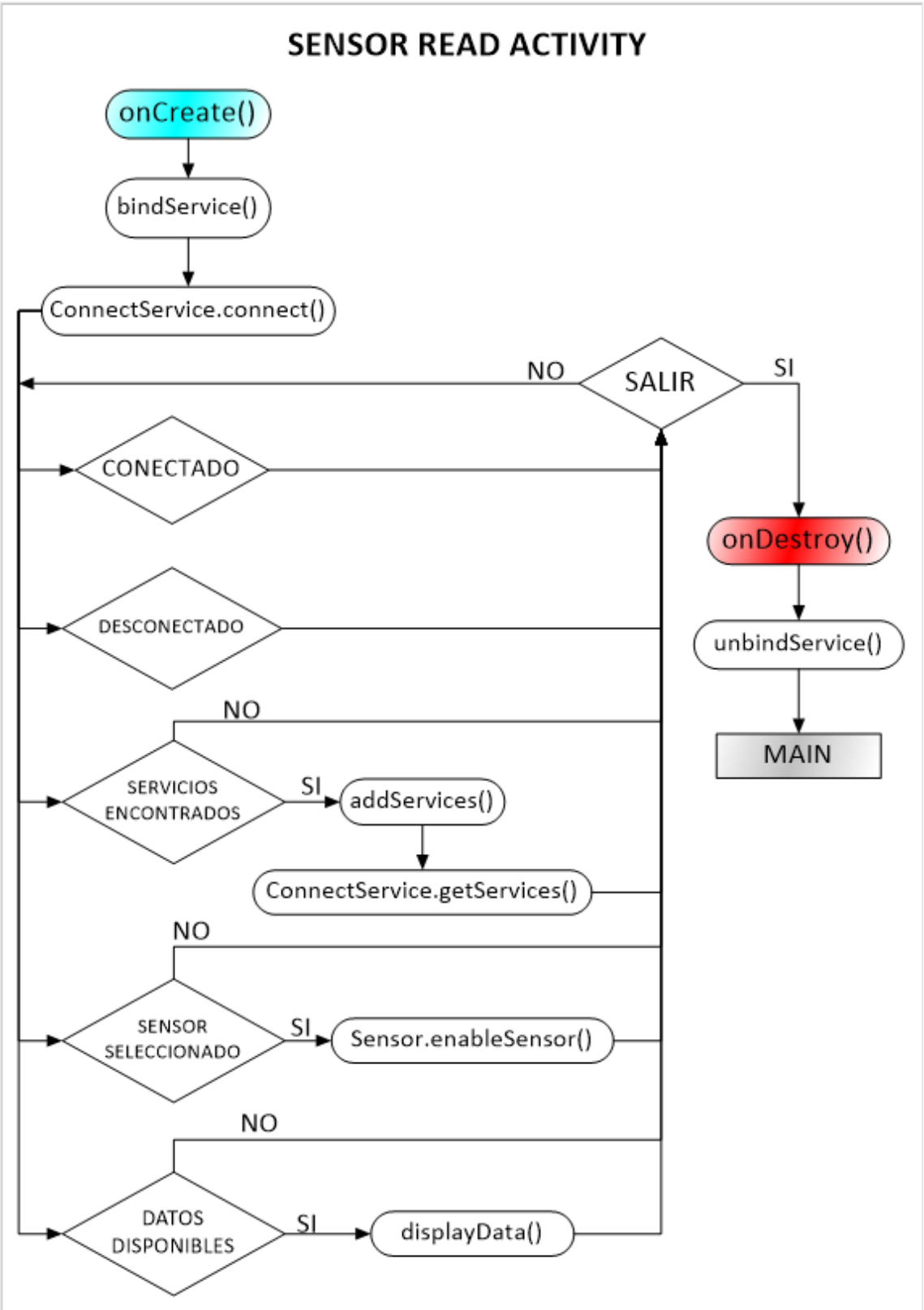

Figura 87. Diagrama de flujo de Sensor Read Activity

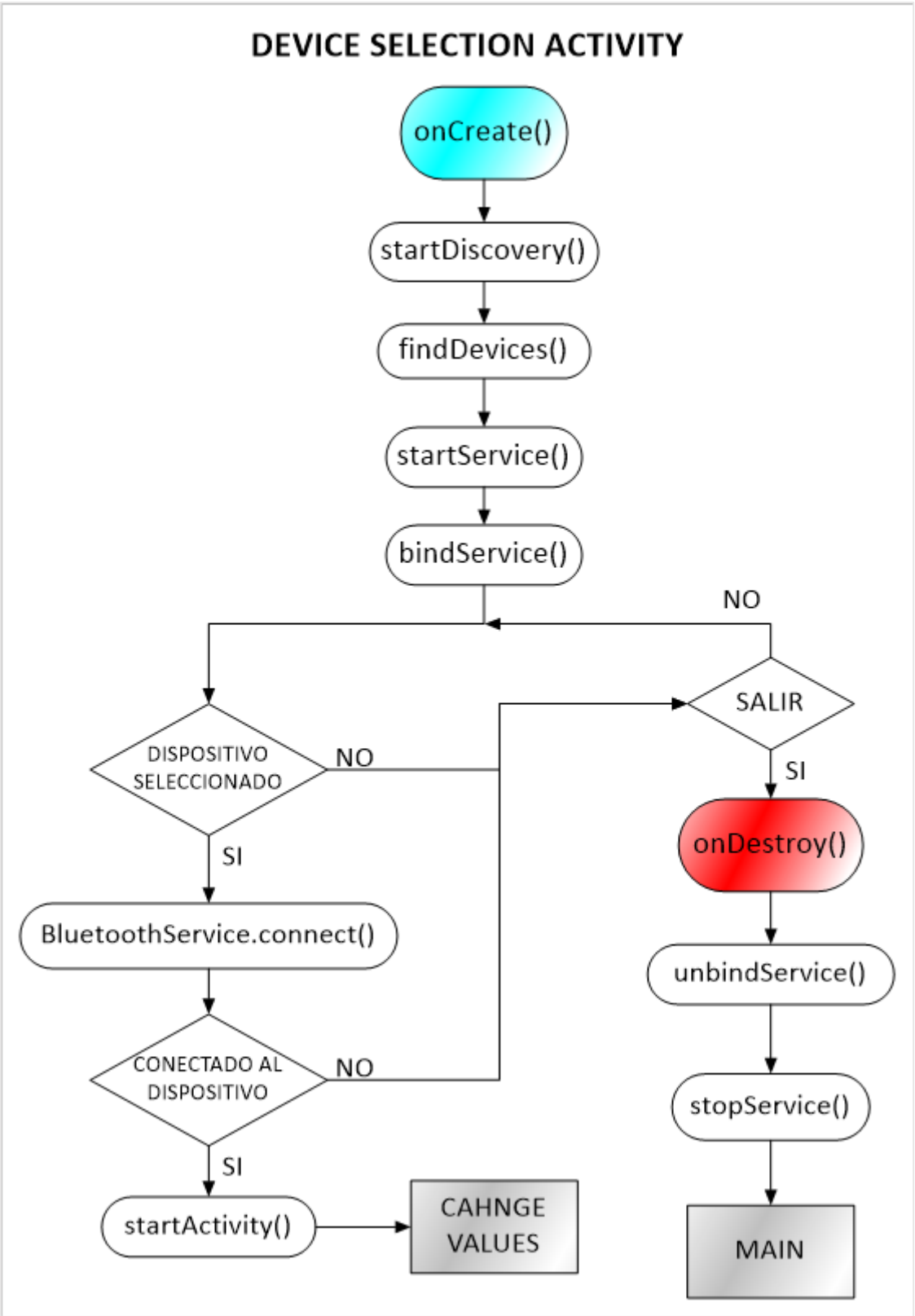

Figura 88. Diagrama de flujo de Device Selection Activity

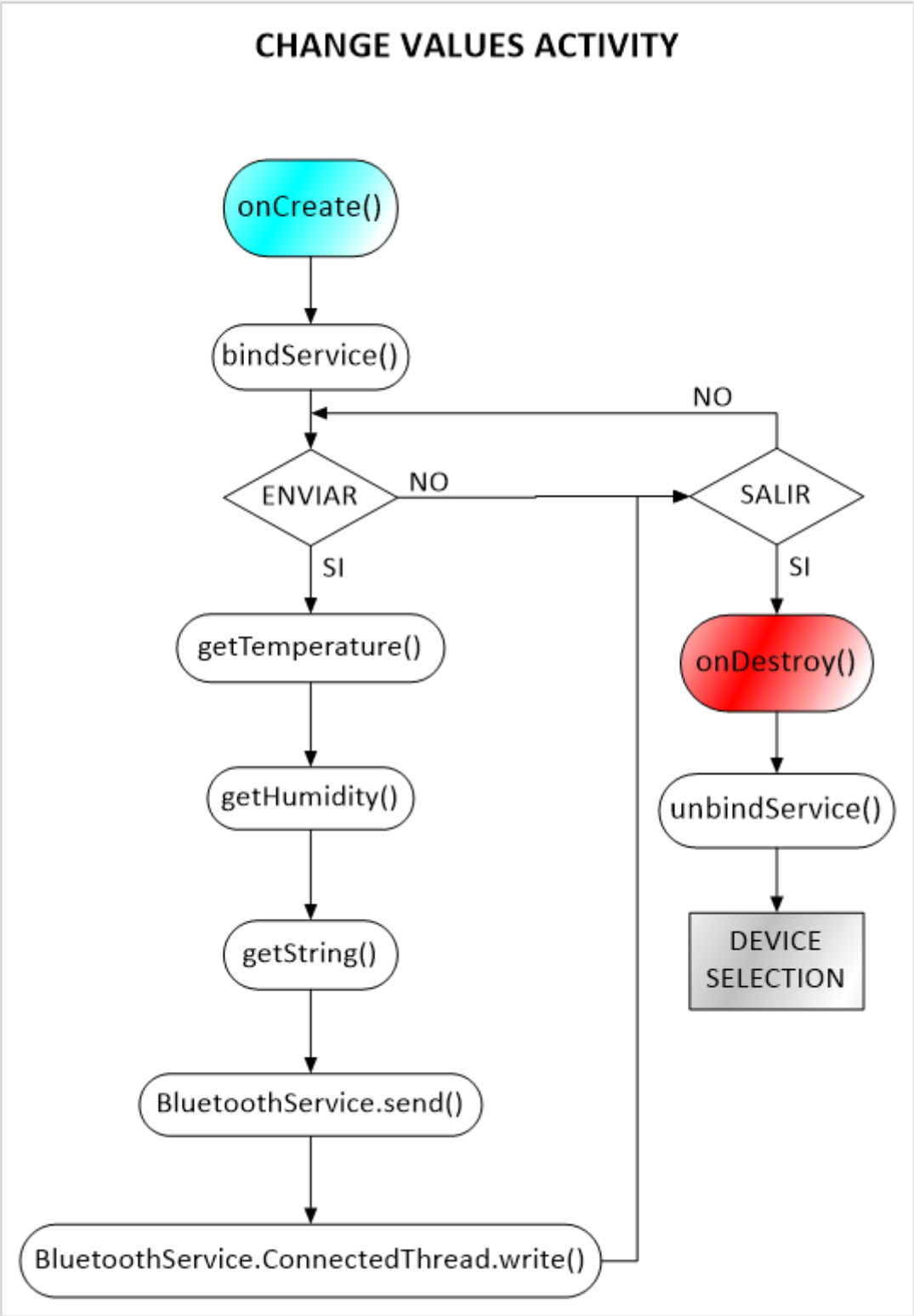

Figura 89. Diagrama de flujo de Change Values Activity

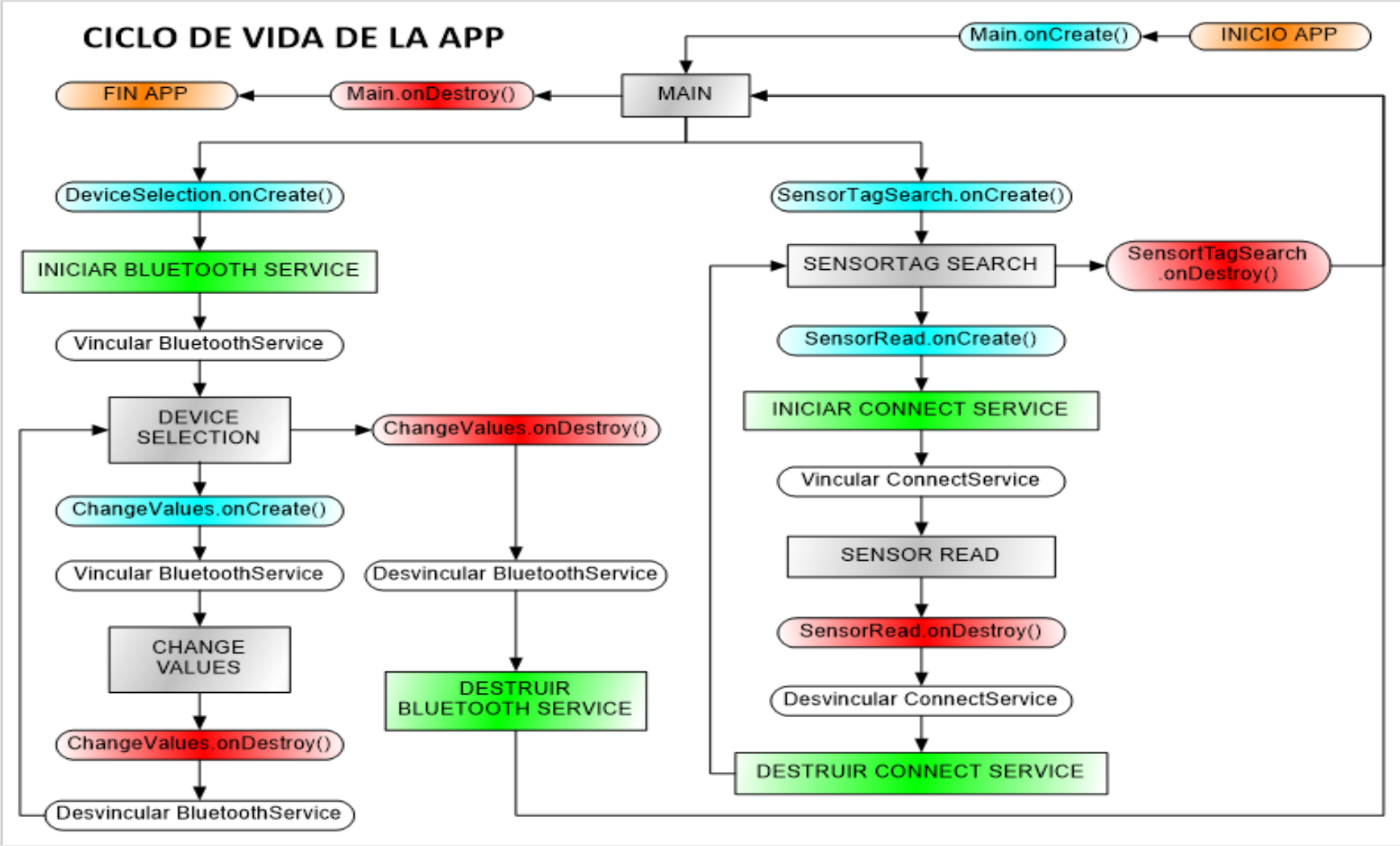

Figura 90. Ciclo de vida de la aplicación